CmdStan Interface

User's Guide

Stan Development Team

CmdStan Version 2.21.0

Friday 18th October, 2019

http://mc-stan.org/

Stan Development Team. 2016. *CmdStan: User's Guide*. Version 2.21.0

Contents

Part I

Introduction

1. Overview

This document is a user's guide for the CmdStan interface to the Stan statistical modeling language. CmdStan takes Stan programs and generates executables that can be run directly from the command line. CmdStan is one of several interfaces to Stan; there are also R, Python, Matlab, Julia, and Stata interfaces.

1.1. Stan Home Page

For links to up-to-date code, examples, manuals, bug reports, feature requests, and everything else Stan related, see the Stan home page:

http://mc-stan.org/

1.2. Licensing

CmdStan, Stan, and the Stan Math Library are licensed under the new BSD license (3-clause). See Appendix A for details, including licensing terms for the dependent packages Boost, Eigen, Sundials, and Intel TBB.

1.3. Modeling Language User's Guide and Reference

Stan's modeling language is shared across all of its interfaces. Stan's language, along with a programming guide and many example models, is detailed in the *Stan Modeling Language User's Guide and Reference Manual*, which is available from the Stan home page (see Section 1.1).

1.4. Example Models

There are many example models for Stan, in addition to those in the user's guide and reference. These are all linked from the Stan home page (see Section 1.1).

1.5. Benefits of CmdStan

Although CmdStan has the least amount of functionality among the Stan interfaces, the minimal nature of CmdStan makes it trivial to install and use the latest development version of the Stan library. It also has the fewest dependencies, which makes it easier to run in limited environments such as clusters. The output generated is in CSV format and can be post-processed using other Stan interfaces or general tools.

2. Getting Started

This chapter is designed to help users get acquainted with the CmdStan interface. Later chapters are devoted to expanding on the material in this chapter with full reference documentation. See the Stan user's manual for details about the Stan language.

2.1. Installation

Installation of CmdStan is simple. CmdStan requires:

- The CmdStan source code and all its libraries.
  This is included in the release tarball or zip file as a single download.
- The `make` utility program.
  This is not strictly necessary, but will make the build process easy. On Windows the `mingw32-make` variant is needed to build CmdStan. This is a `make` variant and can be installed as part of RTools (`https://cran.rstudio.com/bin/windows/Rtools/`). The rest of the documentation assumes `make` (`mingw32-make` on Windows) is available.
- A C++ compiler.

For information about supported versions of Windows, Mac, and Linux platforms see Appendix B. For step-by-step installation instructions of the prerequisites, see

- Windows: Appendix B.3
- Mac: Appendix B.4
- Linux: Appendix B.5.

2.2. Building CmdStan

Building CmdStan involves building two executable programs:

- `stanc`: the Stan compiler (translates Stan language to C++)
- `stansummary`: a basic posterior analysis tool

The build process utilizes the `make` command-line utility and these instructions are applicable for any of our supported platforms.

Steps to build CmdStan:

1. Open a command-line terminal window and change directories to the CmdStan directory. From here on, we'll refer to this location as `<cmdstan-home>`.

   ```
   > cd <cmdstan-home>
   ```

   A listing of the files and directory of this folder should show these files:

   ```
   > ls
   LICENSE             makefile
   README.md           runCmdStanTests.py
   doc                 src
   examples            stan
   make                test-all.sh
   ```

2. *Optional:* Set local `make` variables by editing the file `~/.config/stan/make.local` or `<cmdstan-home>/make/local`. See Appendix B.6 for a list of available options. For most installations, this step can be skipped; the default configuration should work for most users.

3. Use make to build CmdStan. When multiple CPU cores are available on the system, the call to make can be parallelized. It can either be specified directly when calling `make` with the `-jN` option, where `N` is the number of CPU cores. For instance, to run on 4 cores, use

   ```
   > make build -j4
   ```

   Warning: The `make` program may take 10+ minutes and consume 2+ GB of memory to build CmdStan. Please use `mingw32-make` on Windows (available from RTools).

4. *Windows only:* CmdStan requires that the Intel TBB library, which is build by the above command, can be found by the Windows system. This requires that the directory `<cmdstan-home>/stan/lib/stan_math/lib/tbb` is part of the `PATH` environment variable. To permanently make this setting for the current user, you may execute:

   ```
   > mingw32-make install-tbb
   ```

   Don't forget to open a new shell where these new settings will take effect. For other platforms this step is not needed, since the absolute path to the Intel TBB library is linked into Stan programs (not possible on Windows).

When CmdStan is successfully built, the `make` program will report (after other lines of output)

```
--- CmdStan v2.21.0 built ---
```

and there will be two executables in the <cmdstan-home>/bin/ folder:

- stanc, the Stan compiler. The Stan compiler translates a Stan program into C++ code. See Chapter 4 for details.
- stansummary, a posterior analysis tool. The stansummary command summarizes the comma-separated values files that are generated from Stan program runs. For each parameter within the Stan program, stansummary reports the mean, standard deviation, quantiles, $\hat{R}$, and other values. See Chapter 6 for details.

2.3. Compiling and Executing a Stan Program

The rest of this quick-start guide explains how to code and run a very simple Bayesian model.

A Simple Bernoulli Model

The following is a simple, complete Stan program for a Bernoulli model of binary data.[1]

```
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);  // uniform prior on interval 0,1
  y ~ bernoulli(theta);
}
```

The model assumes the binary observed data y[1],...,y[N] are i.i.d. with Bernoulli chance-of-success theta. The prior on theta is beta(1,1) (i.e., uniform).

[1] The model is available with the CmdStan distribution at the path examples/bernoulli/bernoulli.stan.

Data Set

A data set of `N` $= 10$ observations is coded either using JSON notation or in the dump data format created by the `stan_rdump` in package `rstan`. [2] In JSON:

```
{ "N" : 10, "y" : [ 0, 1, 0, 0, 0, 0, 0, 0, 0, 1 ] }
```

In `stan_rdump` format:

```
N <- 10
y <- c(0,1,0,0,0,0,0,0,0,1)
```

This defines the contents of two variables, `N` and `y`, using an R-like syntax (see Chapter D for more information).

Change directories to `<cmdstan-home>`

Before building any Stan program, change directories to `<cmdstan-home>`.

Compiling a Stan Program

A single call to `make` is all that's necessary to translate a Stan program to an executable for the command line. (This call will first translate the Stan program to C++, then compile the C++ code to an executable.)

A Stan program must be in a file with the file extension `.stan`. To create an executable from a Stan program, `make` will be called with the name of the executable as its argument. For Mac and Linux, it is the name of the Stan program with the `.stan` omitted. For Windows, replace `.stan` with `.exe`, and make sure that the path is given with slashes and not backslashes.

To build the Bernoulli example, use the following command for Mac and Linux:

```
> make examples/bernoulli/bernoulli
```

For Windows, the command is the same with the addition of `.exe` at the end of the target (note: use forward slashes):

```
> make examples/bernoulli/bernoulli.exe
```

The generated C++ code (`bernoulli.hpp`) and the compiled executable will be placed in the same directory as the Stan program.

[2] The data is also included with the CmdStan distribution and can be found at path `examples/bernoulli/bernoulli.data.R` and `examples/bernoulli/bernoulli.data.json` .

Note: you must start in the `<cmdstan-home>` directory. The Stan program can be in a different path, but the path to the Stan program must not contain a space. (This is a limitation that's introduced by `make`.) Relative paths are ok; the relative path must not contain a space.

Sampling from the Stan Program

The program can be executed from the directory in which it resides.

```
> cd examples/bernoulli
```

To execute sampling of the model under Linux or Mac, use

```
> ./bernoulli sample data file=bernoulli.data.R
```

In Windows, the `./` prefix is not needed, resulting in the following command.

```
> bernoulli.exe sample data file=bernoulli.data.R
```

The output is the same across all supported platforms. First, the configuration of the program is echoed to the standard output:

```
method = sample (Default)
  sample
    num_samples = 1000 (Default)
    num_warmup = 1000 (Default)
    save_warmup = 0 (Default)
    thin = 1 (Default)
    adapt
      engaged = 1 (Default)
      gamma = 0.050000000000000003 (Default)
      delta = 0.80000000000000004 (Default)
      kappa = 0.75 (Default)
      t0 = 10 (Default)
      init_buffer = 75 (Default)
      term_buffer = 50 (Default)
      window = 25 (Default)
    algorithm = hmc (Default)
      hmc
        engine = nuts (Default)
          nuts
            max_depth = 10 (Default)
        metric = diag_e (Default)
        metric_file =  (Default)
```

```
        stepsize = 1 (Default)
        stepsize_jitter = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 4294967295 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)
```

After the configuration has been displayed, a short timing message is given.

```
Gradient evaluation took 4e-06 seconds
1000 transitions using 10 leapfrog steps per transition would
  take 0.04 seconds.
Adjust your expectations accordingly!
```

Next, the sampler reports the iteration number, reporting the percentage complete.

```
Iteration:    1 / 2000 [  0%]  (Warmup)
Iteration:  100 / 2000 [  5%]  (Warmup)
...
Iteration: 2000 / 2000 [100%]  (Sampling)
```

Sampler Output

Each execution of the model results in draws from a single Markov chain being written to a file in comma-separated value (CSV) format. The default name of the output file is `output.csv`.

The first part of the output file records the version of the underlying Stan library and the configuration as comments (i.e., lines beginning with the pound sign (#)).

```
# stan_version_major = 2
# stan_version_minor = 12
# stan_version_patch = 0
# model = bernoulli_model
# method = sample (Default)
#   sample
#     num_samples = 1000 (Default)
#     num_warmup = 1000 (Default)
#     save_warmup = 0 (Default)
```

```
#     thin = 1 (Default)
...
```

This is followed by a CSV header indicating the names of the values sampled.

```
lp__,accept_stat__,stepsize__,treedepth__,n_leapfrog__,divergent__,energy__,theta
```

The first column reports the unnormalized log probability of the model. The next columns provide sampler-dependent information; here, it is columns two through five. For basic Hamiltonian Monte Carlo (HMC) and its adaptive variant, the No-U-Turn sampler (NUTS), the sampler-dependent parameters are described in the following table.

Sampler	*Parameter*	*Description*
NUTS	`accept_stat__`	Metropolis acceptance probability averaged over samples in the slice
NUTS	`stepsize__`	Integrator step size
NUTS	`treedepth__`	Tree depth
NUTS	`n_leapfrog__`	Number of leapfrog calculations
NUTS	`divergent__`	1 if trajectory diverged
NUTS	`energy__`	Hamiltonian value
HMC	`accept_stat__`	Metropolis acceptance probability
HMC	`stepsize__`	Integrator step size
HMC	`int_time__`	Total integration time
NUTS	`energy__`	Hamiltonian value

The remaining columns correspond to model parameters. For the Bernoulli model, it is just the sixth column, `theta`. The header line is streamed to the output file before warmup begins.

The next section describes the results of adaptation taking place during the warmup phase.

```
# Adaptation terminated
# Step size = 1.81311
# Diagonal elements of inverse mass matrix:
# 0.415719
```

The default sampler is NUTS with an adapted step size and a diagonal inverse mass matrix. For this example, the step size is 1.81311, and the inverse mass contains the single entry 0.415719 corresponding to the parameter `theta`.

Draws from the posterior distribution are printed out next, each line containing a single draw with the columns corresponding to the header. [3]

[3] There are repeated entries due to the Metropolis accept step in the No-U-Turn sampling algorithm.

```
-6.78148,0.958918,0.997192,2,3,0,7.3034,0.283226
-6.74932,0.99923,0.997192,2,3,0,6.77915,0.243658
-6.88104,0.944956,0.997192,2,3,0,7.02671,0.317841
-6.74805,1,0.997192,2,3,0,6.84913,0.249106
-10.0366,0.49441,0.997192,2,3,0,10.1243,0.0398088
...
```

The output ends with timing details,

```
#  Elapsed Time: 0.006811 seconds (Warm-up)
#                0.011645 seconds (Sampling)
#                0.018456 seconds (Total)
```

Summarizing Sampler Output

The command-line program `bin/stansummary` will display summary information about the run (for more information, see Chapter 6). To run `stansummary` on the output file generated for `bernoulli` on Linux or Mac, type

```
> <cmdstan-home>/bin/stansummary output.csv
```

For Windows, use backslashes to call the `stansummary.exe`.

```
> <cmdstan-home>\bin\stansummary.exe output.csv
```

The output of the command will display information about the run followed by information for each parameter and generated quantity. For `bernoulli`, we ran 1 chain and saved 1000 iterations. The information is echoed to the standard output stream. The output is

```
Inference for Stan model: bernoulli_model
1 chains: each with iter=(1000); warmup=(0); thin=(1); 1000 iterations saved.

Warmup took (0.014) seconds, 0.014 seconds total
Sampling took (0.027) seconds, 0.027 seconds total

                Mean     MCSE   StdDev     5%   50%   95%  N_Eff  N_Eff/s    R_hat
lp__            -7.2  3.0e-02  6.6e-01   -8.5  -7.0  -6.7    479    29120  1.0e+00
accept_stat__   0.91  4.5e-03  1.4e-01   0.61  0.97   1.0   1000    60846  1.0e+00
stepsize__      1.00  3.8e-15  2.7e-15   1.00  1.00  1.00   0.50       30  1.0e+00
treedepth__      1.7  1.6e-02  4.7e-01    1.0   2.0   2.0    812    49412  1.0e+00
n_leapfrog__     2.4  3.3e-02  9.5e-01    1.0   3.0   3.0    813    49439  1.0e+00
divergent__     0.00  0.0e+00  0.0e+00   0.00  0.00  0.00   1000    60846      nan
energy__         7.7  4.6e-02  1.0e+00    6.8   7.4   9.6    491    29856  1.0e+00
theta           0.25  6.0e-03  1.1e-01  0.084  0.24  0.46    361    21985  1.0e+00
```

```
Samples were drawn using hmc with nuts.
For each parameter, N_Eff is a crude measure of effective sample size,
and R_hat is the potential scale reduction factor on split chains (at
convergence, R_hat=1).
```

In addition to the general information about the runs, `stansummary` displays summary statistics for each parameter and generated quantity.

In the `bernoulli` model, there is a single parameter, `theta`. The mean, standard error of the mean, standard deviation, the 5%, 50%, and 95% quantiles, number of effective samples (total and per second), and $\hat{R}$ value are displayed. These quantities and their uses are described in detail in the introductory Markov chain Monte Carlo (MCMC) chapter of the language user's guide and reference manual.

The command `bin/stansummary` can be called with more than one csv file by separating filenames with spaces. It will also take wildcards in specifying filenames. A typical usage of Stan from the command line would first create one or more Markov chains by calling the model executable, typically in parallel, writing the output CSV file for each into its own directory. After all of the processes are finished, the results would be analyzed using `stansummary` to assess convergence and inspect the means and quantiles of the fitted variables. Additionally, downstream inferences may be performed using the draws (e.g., to make decisions or predictions for unseen data).

Optimization

CmdStan can be used for finding posterior modes as well as sampling from the posterior distribution. The executable does not need to be recompiled in order to switch from sampling to optimization, and the data input format is the same. The following is a minimal call to Stan's optimizer using defaults for everything but the location of the data file. See Section 9.4 for more details.

```
> ./bernoulli optimize data file=bernoulli.data.R
```

Executing this command prints the following.

```
method = optimize
  optimize
    algorithm = lbfgs (Default)
      lbfgs
        init_alpha = 0.001 (Default)
        tol_obj = 9.9999999999999998e-13 (Default)
        tol_rel_obj = 10000 (Default)
        tol_grad = 1e-08 (Default)
        tol_rel_grad = 10000000 (Default)
        tol_param = 1e-08 (Default)
```

```
        history_size = 5 (Default)
    iter = 2000 (Default)
    save_iterations = 0 (Default)
id = 0 (Default)
data
  file = bernoulli.data.R
init = 2 (Default)
random
  seed = 4294967295 (Default)
output
  file = output.csv (Default)
  diagnostic_file =  (Default)
  refresh = 100 (Default)

initial log joint probability = -5.18908
  Iter     log prob        ||dx||      ||grad||    alpha   alpha0  # evals  Notes
     4     -5.00402    0.00400907   7.80306e-05        1        1        7
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance
```

The first part of the output reports on the configuration used, here indicating the default L-BFGS optimizer, with default initial stepsize and tolerances for monitoring convergence. The second part of the output indicates how well the algorithm fared, here converging and terminating normally. The numbers reported indicate that it took 4 iterations and 7 gradient evaluations, resulting in a final state state where the change in parameters was roughly 0.004 and the length of the gradient roughly 8e-5. The `alpha` value is for step size used. This is, not surprisingly, far fewer iterations than required for sampling; even fewer iterations would be used with less stringent user-specified convergence tolerances.

Optimization Output

The output from optimization is written into the file `output.csv` by default. The output follows the same pattern as the output for sampling, first dumping the entire set of parameters used.

```
# stan_version_major = 2
# stan_version_minor = 7
# stan_version_patch = 0
# model = bernoulli_model
# method = optimize
#   optimize
#     algorithm = lbfgs (Default)
#       lbfgs
#         init_alpha = 0.001 (Default)
```

```
#         tol_obj = 9.9999999999999998e-13 (Default)
#         tol_rel_obj = 10000 (Default)
#         tol_grad = 1e-08 (Default)
#         tol_rel_grad = 10000000 (Default)
#         tol_param = 1e-08 (Default)
#         history_size = 5 (Default)
#     iter = 2000 (Default)
#     save_iterations = 0 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 458923754
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
lp__,theta
-5.00402,0.200008
```

Note that everything is a comment other than a line for the header, and a line for the values. Here, the header indicates the unnormalized log probability with `lp__` and the model parameter `theta`. The maximum log probability is -5.0 and the posterior mode for `theta` is 0.20. The mode exactly matches what we would expect from the data.[4] Because the prior was uniform, the result 0.20 represents the maximum likelihood estimate (MLE) for the very simple Bernoulli model. Note that no uncertainty is reported.

Variational Inference

CmdStan can approximate the posterior distribution using variational inference. The executable does not need to be recompiled in order to switch to variational inference, and the data input format is the same. The following is a minimal call to Stan's variational inference algorithm using defaults for everything but the location of the data file. See Section 9.4 for more details.

```
> ./bernoulli variational data file=bernoulli.data.R
```

Executing this command prints the following.

[4]The Jacobian adjustment included for the sampler's log probability function is not applied during optimization, because it can change the shape of the posterior and hence the solution.

```
 method = variational
   variational
     algorithm = meanfield (Default)
       meanfield
     iter = 10000 (Default)
     grad_samples = 1 (Default)
     elbo_samples = 100 (Default)
     eta = 1 (Default)
     adapt
       engaged = 1 (Default)
       iter = 50 (Default)
     tol_rel_obj = 0.01 (Default)
     eval_elbo = 100 (Default)
     output_samples = 1000 (Default)
 id = 0 (Default)
 data
   file = bernoulli.data.R
 init = 2 (Default)
 random
   seed = 1196271396
 output
   file = output.csv (Default)
   diagnostic_file =  (Default)
   refresh = 100 (Default)

This is Automatic Differentiation Variational Inference.

(EXPERIMENTAL ALGORITHM: expect frequent updates to the procedure.)

Gradient evaluation took 6e-06 seconds
1000 iterations under these settings should take 0.006 seconds.
Adjust your expectations accordingly!

Begin eta adaptation.
Iteration:   1 / 250 [  0%]  (Adaptation)
Iteration:  50 / 250 [ 20%]  (Adaptation)
Iteration: 100 / 250 [ 40%]  (Adaptation)
Iteration: 150 / 250 [ 60%]  (Adaptation)
Iteration: 200 / 250 [ 80%]  (Adaptation)
Success! Found best value [eta = 1] earlier than expected.

Begin stochastic gradient ascent.
  iter       ELBO   delta_ELBO_mean   delta_ELBO_med   notes
   100         -6             1.000            1.000
   200       -6.3             0.511            1.000
   300       -6.2             0.344            0.021
   400       -6.2             0.261            0.021
   500       -6.2             0.211            0.014
   600       -6.2             0.178            0.014
```

```
  700       -6.3            0.154           0.012
  800       -6.3            0.135           0.012
  900       -6.2            0.121           0.012
 1000       -6.4            0.112           0.012
 1100       -6.3            0.015           0.012
 1200       -6.2            0.014           0.012
 1300       -6.1            0.014           0.012
 1400       -6.2            0.015           0.013
 1500       -6.3            0.015           0.012
 1600       -6.3            0.014           0.012
 1700       -6.3            0.013           0.012
 1800       -6.3            0.013           0.012
 1900       -6.2            0.013           0.013
 2000       -6.3            0.011           0.011
 2100       -6.2            0.008           0.009   MEAN ELBO CONVERGED   MEDIAN ELBO

Drawing 1000 samples from the approximate posterior... COMPLETED.
```

The first part of the output reports on the configuration used. Here it indicates the default mean-field setting of the variational inference algorithm. It also indicates the default parameter sizes and tolerances for monitoring the algorithm's convergence. The second part of the output describes the progression of the algorithm. An adaptation phase finds a good value for the step size scaling parameter η. The evidence lower bound (ELBO) is the variational objective function and is evaluated based on a Monte Carlo estimate. The variational inference algorithm in Stan is stochastic, which makes it challenging to assess convergence. That is, while the algorithm appears to have converged in $\sim$100 iterations, the algorithm runs for another few thousand iterations until mean change in ELBO drops below the default tolerance of 0.01.

Variational Inference Output

The output from variational is written into the file `output.csv` by default. The output follows the same pattern as the output for sampling, first dumping the entire set of parameters used.

```
# stan_version_major = 2
# stan_version_minor = 8
# stan_version_patch = 0
# model = bernoulli_model
# method = variational
#   variational
#     algorithm = meanfield (Default)
#       meanfield
```

```
#     iter = 10000 (Default)
#     grad_samples = 1 (Default)
#     elbo_samples = 100 (Default)
#     eta = 1 (Default)
#     adapt
#       engaged = 1 (Default)
#       iter = 50 (Default)
#     tol_rel_obj = 0.01 (Default)
#     eval_elbo = 100 (Default)
#     output_samples = 1000 (Default)
# id = 0 (Default)
# data
#   file = bernoulli.data.R
# init = 2 (Default)
# random
#   seed = 1196271396
# output
#   file = output.csv (Default)
#   diagnostic_file =  (Default)
#   refresh = 100 (Default)
lp__,theta
# Stepsize adaptation complete.
# eta = 1
0,0.249604
0,0.254227
0,0.211049
...
```

Note that everything is a comment other than a line for the header, the adapted value for the stepsize, and a line for the values. The header indicates the unnormalized log probability with `lp__`. This is a legacy feature that we do not use for variational inference. The ELBO is not stored unless a diagnostic option is given. See Section 9.4 for more details. The first line is special: it is the mean of the variational approximation. The rest of the output contains `output_samples` number of samples drawn from the variational approximation.

Configuring Command-Line Options

The command-line options for running a model are detailed in Chapter 9. They can also be printed on the command line using Linux or Mac OS with

```
> ./bernoulli help-all
```

and on Windows with

```
> bernoulli.exe help-all
```

Part II

CmdStan Tools

3. Overview

CmdStan is the command-line interface for Stan. The next two chapters describe tools that are built as part of CmdStan installation: `stanc` and `stansummary`. The process of building a CmdStan executable from a Stan program is as follows:

1. A Stan program is written to file with a `.stan` extension.

2. `stanc` is used to translate the Stan program into a C++ file. This C++ file is not a full program that can be compiled to executable directly, but a translation from the Stan language into a C++ concept. Each interface will generate identical C++ for the same Stan program.

3. A CmdStan executable is generated from the CmdStan source and the generated C++. Each Stan program will have its own CmdStan executable. The options to the CmdStan executable are described in Chapter 9.

3.1. Building the CmdStan Tools

The easy way to build CmdStan is through the use of make. From a command line window, type:

```
> cd <cmdstan-home>
> make build
```

This will build both `stanc` and `stansummary`. If your computer has multiple cores and sufficient ram, the build process can be parallelized by providing the `-j` option. For example, to build on 4 cores, type:

```
> cd <cmdstan-home>
> make -j4 build
```

Warning: The `make` program may take 10+ minutes and consume 2+ GB of memory to build CmdStan.

4. stanc: Translating Stan to C++

4.1. Building the stanc Compiler

Before the stanc compiler can be used, it must be built. It can be compiled directly using the makefile as follows. For Mac and Linux:

```
> make bin/stanc
```

For Windows:

```
> make bin/stanc.exe
```

To change the default compiler or the optimization level, see Appendix B.6.

4.2. The stanc Compiler

The stanc compiler converts Stan programs to C++ concepts. The first stage of compilation involves parsing the text of the Stan program. If the parser is successful, the second stage of compilation generates C++ code. If the parser fails, it will provide an error message indicating the location in the input where the failure occurred and reason for the failure.

The following example illustrates a fully qualified call to stanc to build the simple Bernoulli model.

For Linux and Mac:

```
> cd <cmdstan-home>
> bin/stanc --name=bernoulli --o=bernoulli.hpp \
  examples/bernoulli/bernoulli.stan
```

The backslash (\) is a continuation of the same line and can be omitted if the command is on a single line.

For Windows:

```
> cd <cmdstan-home>
> bin\stanc.exe --name=bernoulli --o=bernoulli.hpp ^
    examples/bernoulli/bernoulli.stan
```

(The caret (^) is a line continuation on Windows.)

This call specifies the name of the model, here bernoulli. This will determine the name of the class implementing the model in the C++ code. Because this name is the name of a C++ class, it must start with an alphabetic character (a-z or A-Z)

and contain only alphanumeric characters (a-z, A-Z, and 0-9) and underscores (_) and should not conflict with any C++ reserved keyword.

The C++ code implementing the class is written to the file bernoulli.hpp in the current directory. The final argument, bernoulli.stan, is the file from which to read the Stan program.

4.3. Command-Line Options for stanc

The model translation program stanc is called as follows.

```
> stanc [options] model_file
```

The argument *model_file* is a path to a Stan model file ending in suffix .stan. The options are as follows.

--help
: Displays the manual page for stanc. If this option is selected, nothing else is done.

--version
: Prints the version of stanc. This is useful for bug reporting and asking for help on the mailing lists.

--name=*class_name*
: Specify the name of the class used for the implementation of the Stan model in the generated C++ code.

  Default: *class_name* = *model_file*_model

--o=*cpp_file_name*
: Specify the name of the file into which the generated C++ is written.

  Default: *cpp_file_name* = *class_name*.hpp

--*allow_undefined*
: Do not throw a parser error if there is a function in the Stan program that is declared but not defined in the functions block.

4.4. Using External C++ Code

The -allow_undefined flag can be passed to the call to stanc, which will allow undefined functions in the Stan language to be parsed without an error. We can then include a definition of the function in a C++ header file. We typically control these

options with two make variables: STANCFLAGS and USER_HEADER. See Appendix B.6 for more details.

The C++ file will not compile unless there is a header file that defines a function with the same name and signature in a namespace that is formed by concatenating the class_name argument to stanc documented above to the string _namespace.

For more details about how to write C++ code using the Stan Math Library, see https://arxiv.org/abs/1509.07164. As an example, consider the following variant of the Bernoulli example

```
functions {
  real make_odds(real theta);
}
data {
  int<lower=0> N;
  int<lower=0,upper=1> y[N];
}
parameters {
  real<lower=0,upper=1> theta;
}
model {
  theta ~ beta(1,1);  // uniform prior on interval 0,1
  y ~ bernoulli(theta);
}
generated quantities {
  real odds;
  odds = make_odds(theta);
}
```

Here the make_odds function is declared but not defined, which would ordinarily result in a parser error. However, if you put STANCFLAGS = --allow_undefined into the make/local file or into the stanc call, then the above Stan program will parse successfully but would not compile when you call

```
> make examples/bernoulli/bernoulli # on Windows add .exe
```

To compile successfully, you need to write a file such as examples/bernoulli/make_odds.hpp with the following lines

```
namespace bernoulli_model_namespace {

  template <typename T0__>
```

```
  inline
  typename boost::math::tools::promote_args<T0__>::type
  make_odds(const T0__& theta, std::ostream* pstream__) {
    return theta / (1 - theta);
  }

}
```

Thus, the following `make` invocation should work

```
> STANCFLAGS=--allow_undefined \
USER_HEADER=examples/bernoulli/make_odds.hpp \
make examples/bernoulli/bernoulli # on Windows add .exe
```

or you could put `STANCFLAGS` and `USER_HEADER` into the `make/local` file instead of specifying them on the command-line.

If the function were more complicated and involved functions in the Stan Math Library, then you would need to prefix the function calls with `stan::math::`. The `pstream__` argument is mandatory in the signature but need not be used if your function does not print any output. To see the necessary boilerplate look at the corresponding lines in the generated C++ file.

5. `print`: Output Analysis (deprecated)

`print` is deprecated, but is still available until CmdStan v3.0. See the next chapter for usage (replace `stansummary` with `print`).

6. `stansummary`: Output Analysis

CmdStan is distributed with a posterior analysis utility that is able to read in the output of one or more Markov chains and summarize the posterior fits. This operation mimics the `print(fit)` command in RStan, which itself was modeled on the print functions from R2WinBUGS and R2jags.

6.1. Building the stansummary Command

CmdStan's `stansummary` command is built along with `stanc` into the `bin` directory. It can be compiled directly using the makefile as follows.

```
> cd <cmdstan-home>
> make bin/stansummary
```

6.2. Running the stansummary Command

The `stansummary` command is executed on one or more `output.csv` files. These files may be provided as command-line arguments separated by spaces. That means that wildcards may be used, as they will be replaced by space-separated file names by the operating system's command-line interpreter.

Suppose there are three samples files in a directory generated by fitting a negative binomial model to a small data set.

```
> ls output*.csv

output1.csv        output2.csv        output3.csv

> bin/stansummary output*.csv
```

The result of `bin/stansummary` is displayed in Figure 6.1.[1] The posterior is skewed to the high side, resulting in posterior means ($\alpha = 17$ and $\beta = 10$) that are a long way away from the posterior medians ($\alpha = 9.5$ and $\beta = 6.2$); the posterior median is the value listed under `50%`, which is the 50th percentile of the posterior values.

For Windows, the forward slash in paths need to be converted to backslashes.

[1] RStan's and PyStan's output analysis stansummary may be different than that in the command-line version of Stan.

```
Inference for Stan model: negative_binomial_model
1 chains: each with iter=(1000); warmup=(0); thin=(1); 1000 iterations saved.

Warmup took (0.054) seconds, 0.054 seconds total
Sampling took (0.059) seconds, 0.059 seconds total

                Mean     MCSE   StdDev    5%   50%   95%  N_Eff  N_Eff/s    R_hat
lp__             -14  7.0e-02  1.1e+00   -17   -14   -13    226     3022  1.0e+00
accept_stat__   0.94  3.1e-03  9.7e-02  0.75  0.98   1.0   1000    13388  1.0e+00
stepsize__      0.16  5.1e-16  3.6e-16  0.16  0.16  0.16   0.50      6.7  1.0e+00
treedepth__      2.9  4.1e-02  1.2e+00   1.0   3.0   5.0    829    11104  1.0e+00
n_leapfrog__     8.0  2.1e-01  6.3e+00   1.0   7.0    19    870    11648  1.0e+00
divergent__     0.00  0.0e+00  0.0e+00  0.00  0.00  0.00   1000    13388      nan
energy__          15  8.7e-02  1.5e+00    14    15    18    282     3775  1.0e+00
alpha             16  1.9e+00  2.0e+01   1.9   9.7    50    114     1524  1.0e+00
beta             9.9  1.1e+00  1.2e+01   1.1   6.1    31    124     1664  1.0e+00

Samples were drawn using hmc with nuts.
For each parameter, N_Eff is a crude measure of effective sample size,
and R_hat is the potential scale reduction factor on split chains (at
convergence, R_hat=1).
```

Figure 6.1: *Example output from* `bin/stansummary`. *The model parameters are* `alpha` *and* `beta`. *The values for each quantity are the posterior means, standard deviations, and quantiles, along with Monte-Carlo standard error, effective sample size estimates (per second), and convergence diagnostic statistic. These values are all estimated from samples. In addition to the parameters,* `bin/stansummary` *also outputs* `lp__`*, the total log probability density (up to an additive constant) at each sample, as well as NUTS-specific values that can be helpful in diagnostics. The quantity* `accept_stat__` *is the average Metropolis acceptance probability over each simulated Hamiltonian trajectory and* `stepsize__` *is the integrator step size used in each simulation.* `treedepth__` *is the depth of tree used by NUTS while* `n_leapfrog__` *is the number of leapfrog steps taken during the Hamiltonian simulation;* `treedepth__` *should always be the binary log of* `n_leapfrog__`. `divergent__` *indicates whether or not the simulated Hamiltonian trajectory became unstable and diverged. Finally,* `energy__` *is value of the Hamiltonian (up to an additive constant) at each sample, also known as the energy.*

Output of stansummary Command

`divergent`

CmdStan uses a symplectic integrator to approximate the exact solution of the Hamiltonian dynamics, and when the step size is too large relative to the curvature of the log posterior this approximation becomes unstable and the trajectories can diverge and threaten the validity of the sampler; `divergent` indicates whether or not a given trajectory diverged. If there are any divergences then the samples

may be biased – common solutions are decreasing the step size (often by increasing the target average acceptance probability) or reparameterizing the model.

`energy`

The energy, `energy`, is used to diagnose the accuracy of any Hamiltonian Monte Carlo sampler. If the standard deviation of `energy` is much larger than $\sqrt{D/2}$, where D is the number of *unconstrained* parameters, then the sampler is unlikely to be able to explore the posterior adequately. This is usually due to heavy-tailed posteriors and can sometime be remedied by reparameterizing the model.

6.3. Command-line Options

In addition to the filenames, `stansummary` includes three flags to customize the output.

`help`
: stansummary usage information
No help output by default

`sig_figs=<int>`
: Sets the number of significant figures displayed in the output
Valid values: 0 <sig_figs
(default = 2)

`autocorr=<int>`
: Calculates and then displays the autocorrelation of the specified chain
Valid values: Any integer matching a chain index
(No autocorrelation output by default)

`csv_file=<string>`
: Writes output as a csv file with comments written as #
Valid values: Any valid filename
(Appends output to the file if it exists)

7. `diagnose`: Diagnosing Biased Hamiltonian Monte Carlo Inferences

CmdStan is distributed with a utility that is able to read in and analyze the output of one or more Markov chains to check for the following potential problems:

- Transitions that hit the maximum treedepth
- Divergent transitions
- Low E-BFMI values
- Low effective sample sizes
- High $\hat{R}$ values

The meanings of several of these problems are discussed in http://mc-stan.org/misc/warnings.html#runtime-warnings and https://arxiv.org/abs/1701.02434.

7.1. Building the `diagnose` Command

CmdStan's `diagnose` command is built along with `stanc` into the `bin` directory. It can be compiled directly using the makefile as follows.

```
> cd <cmdstan-home>
> make bin/diagnose
```

7.2. Running the `diagnose` Command

The `diagnose` command is executed on one or more output files, which are provided as command-line arguments separated by spaces. If there are no apparent problems with the output files passed to `bin/diagnose`, it outputs a message that all transitions are within treedepth limit and that no divergent transitions were found.

It problems are detected, it outputs a summary of the problem along with possible ways to mitigate it. As an example, we use the the "eight schools" model from

Stan's example models and its corresponding data.[1] The model is run with four chains and the random seed 12345, leaving the output files `eight_schools1.csv`, `eight_schools2.csv`, etc. The `diagnose` command is then run as follows:

```
> bin/diagnose eight_schools*.csv
```

The result of `bin/diagnose` is displayed in Figure 7.1, indicating two problems, one with divergent transitions, and one indicating a low E-BFMI, and possible ways to solve these problems. The first problem indicates that the parameter `delta` of the sampling algorithm needs to be increased. Since the contents of `eight_schools1.csv` contains the lines

```
#     adapt
#       engaged = 1 (Default)
#       gamma = 0.050000000000000003 (Default)
#       delta = 0.80000000000000004 (Default)
```

this suggests that `delta` should be increased beyond 0.8. Following section 9.5, this suggests that the model perhaps should be rerun as follows:

```
> for i in {1..4}
  do
    ./eight_schools sample adapt delta=0.9 \
       random seed=12345 id=$i data \
       file=eight_schools.data.R \
       output file=eight_schools$i.csv &
  done
```

The online references (http://mc-stan.org/misc/warnings.html#runtime-warnings and https://arxiv.org/abs/1701.02434) contain suggestions for other diagnostic warning; however the correct resolution is necessarily model specific, hence all suggestions general guidelines only.

[1] The model and associated data files are here:

- https://github.com/stan-dev/example-models/blob/master/misc/eight_schools/eight_schools.stan
- https://github.com/stan-dev/example-models/blob/master/misc/eight_schools/eight_schools.data.R

```
Checking sampler transitions for divergences.
95 of 4000 (2.4%) transitions ended with a divergence.
These divergent transitions indicate that HMC is not fully
able to explore the posterior distribution.
Try increasing adapt delta closer to 1.
If this doesn't remove all divergences, try to reparameterize the model.

Checking E-BFMI - sampler transitions HMC potential energy.
The E-BFMI, 0.27, is below the nominal threshold of 0.3 which suggests
that HMC may have trouble exploring the target distribution.
If possible, try to reparameterize the model.
```

Figure 7.1: Example output from `bin/diagnose`.

Part III

CmdStan Executables

8. Compiling CmdStan Executables

Preparing a Stan program to be run involves two steps,

1. translating the Stan program to C++, and
2. compiling the resulting C++ to an executable.

This chapter discusses both steps, as well as their encapsulation into a single `make` target.

8.1. Translating and Compiling through `make`

The simplest way to compile a CmdStan program is through the `make` build tool, which encapsulates the translation and compilation step into a single command. The commands making up the `make` target for compiling a model are described in the following sections, and the following chapter describes how to run a compiled model.

Before compiling a CmdStan program, change directories to `<cmdstan-home>`.

Translating and Compiling Test Models

There are a number of example models distributed with CmdStan which unpack into the path `examples`. To build the simple example `examples/bernoulli/bernoulli.stan`, the following call to `make` suffices.

The following call will build an executable form of the Bernoulli estimator. On Windows, replace `bernoulli` with `bernoulli.exe`.

```
> make examples/bernoulli/bernoulli
```

This will translate the model `bernoulli.stan` to a C++ file, `bernoulli.hpp`, and compile a CmdStan program using the generated C++ file, putting the executable in `examples/bernoulli/bernoulli(.exe)`.

Stan programs do not need to be in the `<cmdstan-home>` directory. The current limitation is that the target executable name can not have spaces – this includes the path to the executable. Spaces in the full path can be avoided by using relative paths. For Windows users, if using the full path, include the drive letter and use forward slashes, e.g. `make c:/cmdstan/examples/bernoulli/bernoulli.exe`.

Dependencies in `make`

When executing a `make` target, all its dependencies are checked to see if they are up to date, and if they are not, they are rebuilt. If the `make` target to build the Bernoulli estimator is invoked a second time, it will see that it is up to date, and will not recompile the program.

If the file containing the Stan program is updated, the next call to `make` will rebuild the CmdStan executable.

Getting Help from the `makefile`

CmdStan's `makefile`, which contains the top-level instructions to `make`, provides extensive help in terms of targets and options. Invoke `make` with the target `help`:

```
> make help
```

Options to `make`

CmdStan allows users to change compilers, library versions for Boost, Eigen, Sundials, and Intel TBB, as well as compilation options such as optimization.

For a full list of options, see Appendix B.6

Clean Targets

A very useful target is `clean-all`, invoked as

```
> make clean-all
```

This removes the CmdStan tools. This step is necessary when changing compilers or other `make` options.

9. Running a CmdStan Program

Once a CmdStan program is compiled, it can be run in many different ways. It can be used to sample or optimize parameters, or to diagnose a model. Before diving into the detailed configurations, the first section provides some simple examples.

9.1. Getting Started by Example

Once a CmdStan program has been converted to a C++ program for that model (see Chapter 4) and the resulting C++ program compiled to a platform-specific executable (see Chapter 8), the model is ready to be run.

All of the CmdStan functionality is highly configurable from the command line; the options are defined later in this chapter. Each command option also has defaults, which are used in this section.

Sampling

Suppose the executable is in file `my_model` and the data is in file `my_data`, both in the current working directory. To generate samples from a data set using the default settings, use one of the following, depending on platform.

Mac OS and Linux

```
> ./my_model sample data file=my_data
```

Windows

```
> my_model sample data file=my_data
```

On both platforms, this command reads the data from file `my_data`, runs warmup tuning for 1000 iterations (the values of which are discarded), and then runs the fully-adaptive NUTS sampler for 1000 iterations, writing the parameter (and other) values to the file `samples.csv` in the current working directory. When no random number seed is specified, a seed is generated from the system time.

Sampling in Parallel

The previous example executes one chain, which can be repeated to generate multiple chains. However, users may want to execute chains in parallel on a multicore machine.

Mac OS and Linux

To sample four chains using a Bash shell on Mac OS or Linux, execute[1]

```
> for i in {1..4}
  do
    ./my_model sample random seed=12345 \
       id=$i data file=my_data \
       output file=samples$i.csv &
 done
```

The ampersand (`&`) at the end of the nested command pushes each process into the background, so that the loop can continue without waiting for the current chain to finish. The `id` value makes sure that a non-overlapping set of random numbers are used for each chain. Also note that the output file is explicitly specified, with the variable `$i` being used to ensure the output file name for each chain is unique.

The terminal standard output will be interleaved for all chains running concurrently. To suppress all terminal output, direct the standard output to the "null" device. This is achieved by postfixing `> /dev/null` to a command, which in the above case, means changing the second-to-last line to

```
       output file=samples$i.csv > /dev/null &
```

Windows

On Windows, the following is functionally equivalent to the Bash snippet above

```
> for /l %x in (1, 1, 4) do start /b model  sample   ^
   random seed=12345 id=%x data file=my_data         ^
   output file=samples%x.csv
```

The caret (`^`) indicates a line continuation in DOS.

Combining Parallel Chains

CmdStan has commands to analyze the output of multiple chains, each stored in their own file; see Chapter 6. RStan also has commands to read in multiple CSV files produced by CmdStan's command-line sampler.

To compute posterior quantities, it is sometimes easier to have the chains merged into a single CSV file. If the grep and sed programs are installed, then the following will combine the four comma-separated values files into a single comma-separated values file. The command is the same on Windows, Mac OS, and Linux.

[1] Complicated multiline commands such as this one are prime candidates for putting into a script file.

```
> grep lp__ samples1.csv > combined.csv
> sed '/^[#l]/d' samples*.csv >> combined.csv
```

Scripting and Batching

The previous examples show how to sample in parallel from the command line. Operations like these can also be scripted, using shell scripts (`.sh`) on Mac OS and Linux and DOS batch (`.bat`) files on Windows. A sequence of several such commands can be executed from a single script file. Such scripts might contain `stanc` commands (see Chapter 4) and `stansummary` commands (see Chapter 6) can be executed from a single script file. At some point, it is worthwhile to move to something with stronger dependency control such as makefiles.

Optimization

CmdStan can find the posterior mode (assuming there is one). If the posterior is not convex, there is no guarantee Stan will be able to find the global mode as opposed to a local optimum of log probability.

For optimization, the mode is calculated without the Jacobian adjustment for constrained variables, which shifts the mode due to the change of variables. Thus modes correspond to modes of the model as written.

Windows

```
> my_model optimize data file=my_data
```

Mac OS and Linux

```
> ./my_model optimize data file=my_data
```

Variational Inference

CmdStan can fit a variational approximation to the posterior. The approximation is a Gaussian in the unconstrained variable space. Stan implements two variational algorithms. The `algorithm=meanfield` option uses a fully factorized Gaussian for the approximation. The `algorithm=fullrank` option uses a Gaussian with a full-rank covariance matrix for the approximation.

Mac OS and Linux

```
> ./my_model variational algorithm=meanfield  \
              data file=my_data
```

```
> ./my_model variational algorithm=fullrank   \
            data file=my_data
```

Windows

```
> my_model variational algorithm=meanfield    \
           data file=my_data
> my_model variational algorithm=fullrank     \
           data file=my_data
```

9.2. Diagnostics

CmdStan has a basic diagnostic feature that will calculate gradients of the initial state and compare them with those calculated with finite differences. If there are discrepancies, there is a problem with the model or initial states (or a bug in Stan). To run on the different platforms, use one of the following.

Mac OS and Linux

```
> ./my_model diagnose data file=my_data
```

Windows

```
> my_model diagnose data file=my_data
```

9.3. Generate Quantities

CmdStan can be used to generate additional quantities of interest given a model, data, and a sample drawn from the model conditioned on the data. For each draw in the sample, CmdStan runs the `generated quantities` block using the parameter estimates for that draw as the parameter values.

Mac OS and Linux

```
> ./my_model generate_quantities fitted_params=my_samples.csv \
           data file=my_data
```

Windows

```
> my_model generate_quantities fitted_params=my_samples.csv \
           data file=my_data
```

9.4. Command-Line Options

A CmdStan program can be run in many different ways, e.g., sampling, optimization, variational inference. As these different uses require different configuration options, CmdStan has its own syntax for specifying the configuration options in a way that follows the logical dependencies between the Stan program and the way in which is it to be run. The command-line options are specified as a set of top-level set of general configuration categories which take further sets of keyword-value pairs as sub-arguments. The top-level categorical keywords are:

- `method` - required, see below
- `id` - optional, value is non-negative integer, specified as `id=`*`int-value`*
- `random` - optional, single keyword with sub-arguments
- `data` - optional, , single keyword with sub-arguments
- `init` - optional, , single keyword with sub-arguments
- `output` - optional, , single keyword with sub-arguments

CmdStan also provides a `help` option described in the next section.

The `method` argument has a nested set of sub-categories where all sub-arguments are appropriate to the choice of the method argument. Default values are provided for all sub-argument options. All sub-arguments for a given argument can be specified in any order. Because `method` is the only mandatory argument, it can be specified either as `method=`*`method_name`* or simply as *`method_name`*, as seen in previous examples, i.e., the keywords `sample`, `optimize`, `variational`, `diagnose`, or `generate_quantities`.

Help

Informative output can be retrieved either globally, by requesting help at the top-level, or locally, by requesting help deeper into the hierarchy. Note that after any help has been displayed the execution immediately terminates, even if a method has been specified.

If `help` is specified as the only argument then a usage message is displayed. Similarly, specifying `help_all` by itself displays the entire argument hierarchy. Specifying `help` after any argument displays a description and valid options for that argument. For example,

```
./my_model method=sample help
```

provides the top-level options for the `sample` method.

Detailed information on the argument, and all arguments deriving from it, can accessed by specifying `help-all` instead,

```
./my_model method=sample help-all
```

Method

All commands other than `help` must include at least one method, specified explicitly as `method=`*`method_name`* or implicitly with only *`method_name`*. Currently CmdStan supports the following methods:

Method	*Description*
`sample`	sample using MCMC
`optimize`	find posterior mode using optimization
`variational`	fit variational approximation (experimental)
`diagnose`	diagnose models
`generate_quantities`	generate quantities of interest

9.5. Full Argument Hierarchy

Here we present the full argument hierarchy, along with relevant details. Some typical use-case examples are provided in the next section.

Typographical Conventions

The following typographical conventions are obeyed in the hierarchy.

- `arg=`*`<value-type>`*
  Arguments with values; displays the value type, legal values, and default value
- `arg`
  Isolated categorical arguments; displays all valid subarguments
- *`value`*
  Values; describes effect of selecting the value
- `avalue`
  Categorical arguments that appear as values to other arguments; displays all valid subarguments

Top-Level Method Argument

Every command must have exactly one method specified. The value type of `list element` means that the valid values are enumerated as a list.

`method=`*`<list element>`*
: Analysis method (Note that `method=` is optional)
Valid values: `sample, optimize, variational, diagnose, generate_quantities`
(Defaults to `sample`)

Sampling-Specific Arguments

The following arguments are specific to sampling. The method argument `sample` (or `method=sample`) must come first in order to enable the subsequent arguments. The other arguments are optional and may appear in any order.

↳ `sample`
: Bayesian inference with Markov Chain Monte Carlo
Valid subarguments: `num_samples, num_warmup, save_warmup, thin, adapt, algorithm`

↳ ↳ `num_samples=`*`<int>`*
: Number of sampling iterations
Valid values: $0 \leq$ `num_samples`
(Defaults to `1000`)

↳ ↳ `num_warmup=`*`<int>`*
: Number of warmup iterations
Valid values: $0 \leq$ `warmup`
(Defaults to `1000`)

↳ ↳ `save_warmup=`*`<boolean>`*
: Stream warmup samples to output?
Valid values: `0, 1`
(Defaults to `0`)

↳ ↳ `thin=`*`<int>`*
: Period between saved samples
Valid values: $0 <$ `thin`
(Defaults to `1`)

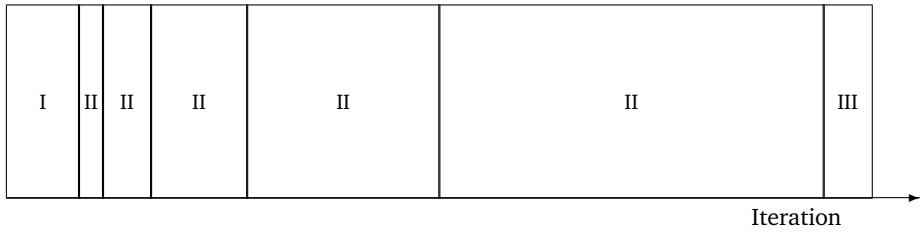

Figure 9.1: *Adaptation during warmup occurs in three stages: an initial fast adaptation interval (I), a series of expanding slow adaptation intervals (II), and a final fast adaptation interval (III). For HMC, both the fast and slow intervals are used for adapting the step size, while the slow intervals are used for learning the (co)variance necessitated by the metric. Iteration numbering starts at 1 on the left side of the figure and increases to the right.*

Sampling Adaptation-Specific Parameters

When adaptation is engaged the warmup period is split into three stages (Figure 9.1), with two *fast* intervals surrounding a series of growing *slow* intervals. Here fast and slow refer to parameters that adapt using local and global information, respectively; the Hamiltonian Monte Carlo samplers, for example, define the step size as a fast parameter and the (co)variance as a slow parameter. The size of the the initial and final fast intervals and the initial size of the slow interval are all customizable, although user-specified values may be modified slightly in order to ensure alignment with the warmup period.

The motivation behind this partitioning of the warmup period is to allow for more robust adaptation. In the initial fast interval the chain is allowed to converge towards the typical set,[2] with only parameters that can learn from local information adapted. After this initial stage parameters that require global information, for example (co)variances, are estimated in a series of expanding, memoryless windows; often fast parameters will be adapted here as well. Lastly the fast parameters are allowed to adapt to the final update of the slow parameters.

Currently all Stan sampling algorithms utilize dual averaging to optimize the step size (this optimization during adaptation of the sampler should not be confused with running Stan's optimization method). This optimization procedure is extremely flexible and for completeness we have exposed each option, using the notation of (Hoffman and Gelman, 2011, 2014). In practice the efficacy of the optimization is sensitive to the value of these parameters, and we do not recommend changing the defaults without experience with the dual averaging algorithm. For

[2]The typical set is a concept borrowed from information theory and refers to the neighborhood (or neighborhoods in multimodal models) of significant posterior probability mass through which the Markov chain will travel in equilibrium.

more information, see the discussion of dual averaging in (Hoffman and Gelman, 2011, 2014).

Variances or covariances are estimated using Welford accumulators to avoid a loss of precision over many floating point operations.

The following subarguments are introduced by the categorical argument `adapt`. Each subargument must contiguously follow `adapt`, though they may appear in any order.

↳ ↳ `adapt`
Warmup Adaptation
Valid subarguments: `engaged, gamma, delta, kappa, t0`

↳ ↳ ↳ `engaged=<boolean>`
Adaptation engaged?
Valid values: `0, 1`
(Defaults to `1`)

↳ ↳ ↳ `gamma=<double>`
Adaptation regularization scale
Valid values: $0 <$ `gamma`
(Defaults to `0.05`)

↳ ↳ ↳ `delta=<double>`
Adaptation target acceptance statistic
Valid values: $0 <$ `delta` < 1
(Defaults to `0.8`)

↳ ↳ ↳ `kappa=<double>`
Adaptation relaxation exponent
Valid values: $0 <$ `kappa`
(Defaults to `0.75`)

↳ ↳ ↳ `t0=<double>`
Adaptation iteration offset
Valid values: $0 <$ `t0`
(Defaults to `10`)

↳ ↳ ↳ `init_buffer=<unsigned int>`
Width of initial fast adaptation interval
Valid values: All
(Defaults to `75`)

↳ ↳ ↳ `term_buffer=<unsigned int>`
Width of final fast adaptation interval
Valid values: All
(Defaults to 50)

↳ ↳ ↳ `window=<unsigned int>`
Initial width of slow adaptation interval
Valid values: All
(Defaults to 25)

By setting the acceptance statistic `delta` to a value closer to 1 (its value must be strictly less than 1 and its default value is 0.8), adaptation will be forced to use smaller step sizes. This can improve sampling efficiency (effective samples per iteration) at the cost of increased iteration times. Raising the value of `delta` will also allow some models that would otherwise get stuck overcome their blockages; see also the `stepsize_jitter` argument.

Sampling Algorithm- and Engine-Specific Arguments

The following batch of arguments are used to control the sampler used for sampling. The top-level specification is for engine, the only valid value of which is `hmc` (this will change in the future as we add new samplers).

↳ ↳ `algorithm=<list element>`
Sampling algorithm
Valid values: `hmc`, `fixed_param`
(Defaults to `hmc`)

Hamiltonian Monte Carlo is a very general approach to sampling that utilizes techniques of differential geometry and mathematical physics to generate efficient MCMC transitions. This generality manifests in a wealth of implementation choices.

↳ ↳ ↳ `hmc`
Hamiltonian Monte Carlo
Valid subarguments: `engine, metric, stepsize, stepsize_jitter`

All HMC implementations require at least two parameters: an integration step size and a total integration time. We refer to different specifications of the latter as *engines*.

In the `static_hmc` implementation the total integration time must be specified by the user, where as the `nuts` implementation uses the No-U-Turn Sampler to determine an optimal integration time dynamically.

↳ ↳ ↳ ↳ engine=<*list element*>
Engine for Hamiltonian Monte Carlo
Valid values: `static, nuts`
(Defaults to `nuts`)

The following options are activated for static HMC.

↳ ↳ ↳ ↳ ↳ `static`
Static integration time
Valid subarguments: `int_time`

↳ ↳ ↳ ↳ ↳ ↳ `int_time`=<*double*>
Total integration time for Hamiltonian evolution
Valid values: $0 <$ `int_time`
(Defaults to 2π)

These options are for NUTS, an adaptive version of HMC.

↳ ↳ ↳ ↳ ↳ `nuts`
The No-U-Turn Sampler
Valid subarguments: `max_depth`

Tree Depth

NUTS generates a proposal by evolving the initial system both forwards and backwards in time to form a balanced binary tree. At each iteration of the NUTS algorithm the tree depth is increased by one, doubling the number of leapfrog steps and effectively doubles the computation time. The algorithm terminates in one of two ways: either the NUTS criterion is satisfied for a new subtree or the completed tree, or the depth of the completed tree hits `max_depth`.

Both the tree depth and the actual number of leapfrog steps computed are reported along with the parameters in the output as `treedepth__` and `n_leapfrog__`, respectively. Because the final subtree may only be partially constructed, these two will always satisfy

$$2^{\text{treedepth}-1} - 1 < N_{\text{leapfrog}} \leq 2^{\text{treedepth}} - 1.$$

`treedepth__` is an important diagnostic tool for NUTS. For example, `treedepth__ = 0` occurs when the first leapfrog step is immediately rejected and the initial state returned, indicating extreme curvature and poorly-chosen step size (at least relative to the current position). On the other hand, if `treedepth__ = max_depth` then NUTS is taking many leapfrog steps and being terminated prematurely to avoid excessively long execution time. For the most efficient sampling

max_depth should be increased to ensure that the NUTS tree can grow as large as necessary.

For more information on the NUTS algorithm see (Hoffman and Gelman, 2011, 2014).

↳ ↳ ↳ ↳ ↳ ↳ ↳ max_depth=<*int*>
Maximum tree depth
Valid values: $0 <$ max_depth
(Defaults to 10)

Euclidean Metric

All HMC implementations in Stan utilize quadratic kinetic energy functions which are specified up to the choice of a symmetric, positive-definite matrix known as a *mass matrix* or, more formally, a *metric* (Betancourt and Stein, 2011).

If the metric is constant then the resulting implementation is known as *Euclidean* HMC. Stan allows for three Euclidean HMC implementations: a unit metric, a diagonal metric, and a dense metric. These can be specified with the values unit_e, diag_e, and dense_e, respectively.

Future versions of Stan will also include dynamic metrics associated with *Riemannian* HMC (Girolami and Calderhead, 2011; Betancourt, 2012).

By default, the metric is estimated during warmup. However, when using a diag_e or dense_e metric, an initial guess for the metric can be specified with the metric_file argument. If provided, the metric_file should contain a single variable, inv_metric, which for a diag_e metric should be a vector of positive values, one for each parameter in the system. For a dense_e metric, inv_metric should be a positive-definite square matrix with number of rows and columns equal to the number of parameters in the model. The file pointed at by metric_file should use either JSON formate or the data dump format (the same format the input data uses). For examples of specifying a vector in the data dump format (as is needed for specifying a diag_e metric), see Section D.3. For examples of specifying a matrix in the data dump format (as is needed for specifying a dense_e metric), see Section D.4.

The metric_file option can be used with and without adaptation enabled.

If adaptation is enabled, the provided metric will be used as the initial guess in the adaptation process. If the initial guess is good, then adaptation should not change it much. If the metric is no good, then the adaptation will override the initial guess. If adaptation is enabled, num_warmup must be set to a value greater than zero. An example of running the sampler with adaptation enabled without specifying a stepsize is

```
> ./model method=sample algorithm=hmc  \
          metric_file=model.metric.data.R  \
          data file=model.data.R init=model.init.R
```

If adaptation is disabled, both a `metric_file` and `stepsize` should be provided to the sampler. Disabling adaptation disables both metric and stepsize adaptation, so a stepsize should be provided along with a metric to enable efficient sampling. An example of providing a `metric_file` and `stepsize` with adaptation disabled is

```
> ./model method=sample adapt engaged=0 algorithm=hmc  \
          metric_file=model.metric.data.R  \
          stepsize=0.1 data file=model.data.R  \
          init=model.init.R
```

↳ ↳ ↳ ↳ `metric=<list element>`
Geometry of base manifold
Valid values: `unit_e, diag_e, dense_e`
(Defaults to `diag_e`)

↳ ↳ ↳ ↳ ↳ *unit_e*
Euclidean manifold with unit metric

↳ ↳ ↳ ↳ ↳ *diag_e*
Euclidean manifold with diag metric

↳ ↳ ↳ ↳ ↳ *dense_e*
Euclidean manifold with dense metric

↳ ↳ ↳ ↳ `metric_file=<string>`
Input file with precomputed Euclidean metric
Valid values: Valid path
(Defaults to empty path)

Step Size and Jitter

All implementations of HMC also use numerical integrators requiring a step size. We also allow that step size to be "jittered" randomly during sampling to avoid any poor interactions with a fixed step size and regions of high curvature. The maximum amount of jitter is 1, which will cause step sizes to be selected in the range of 0 to twice the adapted step size. Low step sizes can get HMC samplers unstuck that would otherwise get stuck with higher step sizes. The downside is that jittering below the adapted value will increase the number of leapfrog steps

required and thus slow down iterations, whereas jittering above the adapted value can cause premature rejection due to simulation error in the Hamiltonian dynamics calculation. See (Neal, 2011) for further discussion of step-size jittering.

↳ ↳ ↳ ↳ `stepsize=<double>`
Step size for discrete evolution
Valid values: $0 <$ `stepsize`
(Defaults to `1`)

↳ ↳ ↳ ↳ `stepsize_jitter=<double>`
Uniformly random jitter of the stepsize, in percent
Valid values: $0 \leq$ `stepsize_jitter` ≤ 1
(Defaults to `0`)

Fixed Parameter Sampler

The fixed parameter sampler generates a new sample without changing the current state of the Markov chain; only generated quantities may change. This can be useful when, for example, trying to generate pseudo-data using the generated quantities block. If the parameters block is empty (no parameters) then using `algorithm=fixed_param` is mandatory.

↳ ↳ ↳ `fixed_param`
Fixed Parameter Sampler

Optimization-Specific Commands

The following arguments are for the top-level method `optimize`. They allow control of the optimization algorithm, and some of its configuration. The other arguments may appear in any order.

↳ `optimize`
Point estimation
Valid subarguments: `algorithm, iter, save_iterations`

↳ ↳ `algorithm=<list element>`
Optimization algorithm
Valid values: `bfgs, lbfgs, newton`
(Defaults to `lbfgs`)

The following options are for the (L-)BFGS optimizer. L-BFGS is the default optimizer and also much faster than the other optimizers.

Convergence monitoring in (L-)BFGS is controlled by a number of tolerance values, any one of which being satisfied causes the algorithm to terminate with a solution.

- The log probability is considered to have converged if

$$|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)| < \texttt{tol\_obj}$$

  or

$$\frac{|\log p(\theta_i|y) - \log p(\theta_{i-1}|y)|}{\max\left(|\log p(\theta_i|y)|, |\log p(\theta_{i-1}|y)|, 1.0\right)} < \texttt{tol\_rel\_obj} * \epsilon.$$

- The parameters are considered to have converged if

$$||\theta_i - \theta_{i-1}|| < \texttt{tol\_param}.$$

- The gradient is considered to have converged to 0 if

$$||g_i|| < \texttt{tol\_grad}$$

  or

$$\frac{g_i^T \hat{H}_i^{-1} g_i}{\max\left(|\log p(\theta_i|y)|, 1.0\right)} < \texttt{tol\_rel\_grad} * \epsilon.$$

Here, i is the current iteration, θ_i is the value of the parameters at iteration i, y is the data, $p(\theta_i|y)$ is the posterior probability of θ_i up to a proportion, ∇_θ is the gradient operator with respect to θ, $g_i = \nabla_\theta \log p(\theta_i|y)$ is the gradient at iteration i, $\hat{H}_i$ is the estimate of the Hessian at iteration i, $|u|$ is absolute value (L1 norm) of u, $||u||$ is vector length (L2 norm) of u, and $\epsilon \approx 2e-16$ is machine precision. Any of the convergence tests can be disabled by setting its corresponding tolerance parameter to zero.

The other command-line argument for (L-)BFGS is `init_alpha`, which is the first step size to try on the initial iteration. If the first iteration takes a long time (and requires a lot of function evaluations), set `init_alpha` to be the roughly equal to the alpha used in that first iteration. `init_alpha` has a tiny default value, which is reasonable for many problems but might be too large or too small depending on the objective function and initialization. Being too big or too small just means that the first iteration will take longer (i.e., require more gradient evaluations) before the line search finds a good step length. It's not a critical parameter, but for optimizing the same model multiple times (as you tweak things or with different data) being able to change it can save some real time.

Finally, L-BFGS has a additional command-line argument, `history_size`, which controls how much memory is used maintaining the approximation of the Hessian. This should be less than the dimensionality of the parameter space and, in general, relatively small values (5 - 10) are sufficient. If L-BFGS performs badly but BFGS is performing well, then consider increasing this. Note that increasing this will increase the memory usage, although this is unlikely to be an issue for typical Stan models.

↳ ↳ ↳ `(l)bfgs`
(L-)BFGS with linesearch
Valid subarguments: `init_alpha`, `tol_obj`, `tol_rel_obj`, `tol_grad`, `tol_rel_grad`, `tol_param`, `history_size` (lbfgs only)

↳ ↳ ↳ ↳ `init_alpha=`*<double>*
Line search step size for first iteration
Valid values: $0 \leq$ `init_alpha`
(Defaults to `0.001`)

↳ ↳ ↳ ↳ `tol_obj=`*<double>*
Convergence tolerance on changes in objective function value
Valid values: $0 \leq$ `tol_obj`
(Defaults to `1e-12`)

↳ ↳ ↳ ↳ `tol_rel_obj=`*<double>*
Convergence tolerance on relative changes in objective function value
Valid values: $0 \leq$ `tol_rel_obj`
(Defaults to `1e+4`)

↳ ↳ ↳ ↳ `tol_grad=`*<double>*
Convergence tolerance on the norm of the gradient
Valid values: $0 \leq$ `tol_grad`
(Defaults to `1e-8`)

↳ ↳ ↳ ↳ `tol_rel_grad=`*<double>*
Convergence tolerance on the relative norm of the gradient
Valid values: $0 \leq$ `tol_rel_grad`
(Defaults to `1e+7`)

↳ ↳ ↳ ↳ `tol_param=`*<double>*
Convergence tolerance on changes in parameter value
Valid values: $0 \leq$ `tol_param`
(Defaults to `1e-8`)

↳ ↳ ↳ ↳ history_size=<*int*>
Number of update vectors to use in Hessian approximations (lbfgs only)
Valid values: $0 <$ history_size
(Defaults to 5)

The following argument is for Newton's optimization method; there are currently no configuration parameters for Newton's method, and it is not recommended because of the slow Hessian calculation involving finite differences.

↳ ↳ ↳ newton
Newton's method

The remaining arguments apply to all optimizers.

↳ ↳ iter=<*int*>
Total number of iterations
Valid values: $0 <$ iter
(Defaults to 2000)

↳ ↳ save_iterations=<*boolean*>
Stream optimization progress to output?
Valid values: 0, 1
(Defaults to 0)

Variational Inference-Specific Commands

The following arguments are for the top-level method variational. They allow control of the variational inference algorithm, and some of its configuration.

↳ variational
Variational inference
Valid subarguments: algorithm, iter, grad_samples, elbo_samples, eta, adapt, tol_rel_obj, eval_elbo, output_samples

↳ ↳ algorithm=<*list element*>
Variational inference algorithm
Valid values: meanfield, fullrank
(Defaults to meanfield)

↳ ↳ iter=<*int*>
Maximum number of iterations
Valid values: $0 <$ iter
(Defaults to 10000)

↳ ↳ `grad_samples=<int>`
Number of samples for Monte Carlo estimate of gradients
Valid values: $0 <$ `grad_samples`
(Defaults to `1`)

↳ ↳ `elbo_samples=<int>`
Number of samples for Monte Carlo estimate of ELBO (objective function)
Valid values: $0 <$ `elbo_samples`
(Defaults to `100`)

↳ ↳ `eta=<double>`
Stepsize weighting parameter for adaptive stepsize sequence
Valid values: $0 <$ `eta`
(Defaults to `1.0`)

↳ ↳ `adapt`
Warmup Adaptation
Valid subarguments: `engaged, iter`

↳ ↳ ↳ `engaged=<boolean>`
Adaptation engaged?
Valid values: `0, 1`
(Defaults to `1`)

↳ ↳ ↳ `iter=<int>`
Maximum number of adaptation iterations
Valid values: $0 <$ `iter`
(Defaults to `50`)

↳ ↳ `tol_rel_obj=<double>`
Convergence tolerance on the relative norm of the objective
Valid values: $0 <$ `tol_rel_obj`
(Defaults to `0.01`)

↳ ↳ `eval_elbo=<int>`
Evaluate ELBO every Nth iteration
Valid values: $0 <$ `eval_elbo`
(Defaults to `100`)

↳ ↳ `output_samples=<int>`
Number of posterior samples to draw and save
Valid values: $0 <$ `output_samples`
(Defaults to `1000`)

Diagnostic-Specific Arguments

The following arguments are specific to diagnostics. As of now, the only diagnostic is gradients of the log probability function.

↳ `diagnose`
  Model diagnostics
  Valid subarguments: `test`

↳ ↳ `test=<list element>`
  Diagnostic test
  Valid values: `gradient`
  (Defaults to `gradient`)

↳ ↳ ↳ *gradient*
  Check model gradient against finite differences Valid subarguments: `epsilon`, `error`

↳ ↳ ↳ ↳ `epsilon=<real>`
  Finite difference step size
  Valid values: $0 <$ `epsilon`
  (Defaults to `1e-6`)

↳ ↳ ↳ ↳ `error=<real>`
  Error threshold
  Valid values: $0 <$ `error`
  (Defaults to `1e-6`)

Generate_quantities-Specific Arguments

The following arguments are for the top-level method `generate_quantities`. They specify the parameter values used to generate additional quantities of interest from the model conditioned on the data.

↳ `generate_quantities`
  Generate quantities of interest using sample of fitted parameter values
  Valid subarguments: `fitted_params`

↳ ↳ `fitted_params=<string>`
  Input file of sample of fitted parameter values for model conditioned on data
  Valid values: Path to existing file
  (Defaults to empty path)

General-Purpose Arguments

The following arguments may be used with any of the previous configurations. They may come either before or after the other subarguments of the top-level method.

Process Identifier Argument

`id=<int>`
: Unique process identifier, used to advance random number generator so that random numbers do not overlap across chains
Valid values: $0 <$ `id`
(Defaults to `0`)

Input Data Arguments

`data`
: Input data options
Valid subarguments: `file`

↳ `file=<string>`
: Input data file
Valid values: Path to existing file
(Defaults to empty path)

Initialization Arguments

Initialization is only applied to parameters defined in the parameters block. Any initial values supplied for transformed parameters or generated quantities are ignored.

`init=<string>`
: Initialization method:
- real number $x > 0$ initializes randomly between [-x, x];
- `0` initializes to 0;
- non-number interpreted as a data file

Valid values: All
(Defaults to `2`)

Random Number Generator Arguments

`random`
: Random number configuration
Valid subarguments: `seed`

↳ `seed=<unsigned int>`
Random number generator seed
Valid values:
- `seed` ≥ 0 generates seed;
- `seed` < 0 uses seed generated from time

(Defaults to `-1`)

Output Arguments

`output`
File output options
Valid subarguments: `file, diagnostic_file, refresh`

↳ `file=<string>`
Output file
Valid values: Valid path
(Defaults to `output.csv`)

↳ `diagnostic_file=<string>`
Auxiliary output file for diagnostic information
Valid values: Valid path
(Defaults to empty path)

↳ `refresh=<int>`
Number of iterations between screen updates
Valid values: $0 <$ `refresh`
(Defaults to `100`)

All Stan arguments have default values, except for the method. This is the only argument that must be specified by the user and a model will not run without it (not to say that the model will run without error, for example a model that requires data will eventually fail unless an input file is specified with `file` under `data`). Assuming that we want to draw MCMC samples from our model, we can either specify a method implicitly,

```
> ./model sample data file=model.data.R init=model.init.R
```

or explicitly,

```
> ./model method=sample data file=model.data.R  \
          init=model.init.R
```

In either case our model now executes without any problem.

Now let's say that we want to customize our execution. In particular we want to set the seed for the random number generator, but we forgot the specific argument syntax. Information for each argument can displayed by calling `help`,

```
> ./model random help
```

which returns

```
random
  Random number configuration
  Valid subarguments: seed
...
```

before printing usage information. For information on the seed argument we just call help one level deeper,

```
> ./model random seed help
```

which returns

```
seed=<unsigned int>
  Random number generator seed
  Valid values: seed > 0, if negative seed is generated from time
  Defaults to -1
  ...
```

Fully informed, we can now run with a given seed,

```
> ./model method=sample data fle=model.data.R  \
          init=model.init.R                    \
          random seed=5
```

The arguments `method`, `data`, `init`, and `random` are all top-level arguments. To really see the power of a hierarchical argument structure let's try to drill down and specify the metric we use for HMC: instead of the default diagonal Euclidean metric, we want to use a dense Euclidean metric. Attempting to specify the metric we try

```
> ./model method=sample data file=model.data.R  \
          init=model.init.R                     \
          random seed=5                         \
          metric=unit
```

only to have the execution fail with the message

```
metric=unit_e is either mistyped or misplaced.
Perhaps you meant one of the following valid configurations?
  method=sample algorithm=hmc metric=<list_element>
Failed to parse arguments, terminating Stan
```

The argument `metric` does exist, but not at the top-level. In order to specify it we have to drill down into sample by first specifying the sampling algorithm, as noted in the suggestion,

```
> ./model method=sample algorithm=hmc metric=unit \
          data file=model.data.R                   \
          init=model.init.R                        \
          random seed=5
```

Unfortunately we still messed up,

```
unit is not a valid value for "metric"
  Valid values: unit_e, diag_e, dense_e
Failed to parse arguments, terminating Stan
```

Tweaking the metric name we make one last attempt,

```
> ./model method=sample algorithm=hmc metric=unit_e \
          data file=model.data.R                    \
          init=model.init.R                         \
          random seed=5
```

which successfully runs.

Finally, let's consider the circumstance where our model runs fine but the NUTS iterations keep saturating the default tree depth limit of 10. We need to change the limit, but how do we specify NUTS let alone the maximum tree depth? To see how let's take advantage of the `help-all` option which prints all arguments that derive from the given argument. We know that NUTS is somehow related to sampling, so we try

```
> ./model method=sample help-all
```

which returns the verbose output,

```
sample
  Bayesian inference with Markov Chain Monte Carlo
  Valid subarguments: num_samples, num_warmup,
                      save_warmup, thin, adapt, algorithm

  num_samples=<int>
    Number of sampling iterations
    Valid values: 0 <= num_samples
    Defaults to 1000

  num_warmup=<int>
    Number of warmup iterations
    Valid values: 0 <= warmup
    Defaults to 1000

  save_warmup=<boolean>
    Stream warmup samples to output?
    Valid values: [0, 1]
    Defaults to 0

  thin=<int>
    Period between saved samples
    Valid values: 0 < thin
    Defaults to 1

  adapt
    Warmup Adaptation
    Valid subarguments: engaged, gamma, delta, kappa, t0

    engaged=<boolean>
      Adaptation engaged?
      Valid values: [0, 1]
      Defaults to 1

    gamma=<double>
      Adaptation regularization scale
      Valid values: 0 < gamma
      Defaults to 0.05

    delta=<double>
      Adaptation target acceptance statistic
      Valid values: 0 < delta < 1
      Defaults to 0.65
```

```
  kappa=<double>
    Adaptation relaxation exponent
    Valid values: 0 < kappa
    Defaults to 0.75

  t0=<double>
    Adaptation iteration offset
    Valid values: 0 < t0
    Defaults to 10

algorithm=<list element>
  Sampling algorithm
  Valid values: hmc
  Defaults to hmc

  hmc
    Hamiltonian Monte Carlo
    Valid subarguments: engine, metric, stepsize,
                        stepsize_jitter

    engine=<list element>
      Engine for Hamiltonian Monte Carlo
      Valid values: static, nuts
      Defaults to nuts

      static
        Static integration time
        Valid subarguments: int_time

        int_time=<double>
          Total integration time for Hamiltonian evolution
          Valid values: 0 < int_time
          Defaults to 2 * pi

      nuts
        The No-U-Turn Sampler
        Valid subarguments: max_depth

        max_depth=<int>
          Maximum tree depth
          Valid values: 0 < max_depth
          Defaults to 10
```

```
      metric=<list element>
        Geometry of base manifold
        Valid values: unit_e, diag_e, dense_e
        Defaults to diag_e

        unit_e
          Euclidean manifold with unit metric

        diag_e
          Euclidean manifold with diag metric

        dense_e
          Euclidean manifold with dense metric

      metric_file=<string>
        Input file with precomputed Euclidean metric
        Valid values: Path to existing file
        Defaults to ""

      stepsize=<double>
        Step size for discrete evolution
        Valid values: 0 < stepsize
        Defaults to 1

      stepsize_jitter=<double>
        Uniformly random jitter of the stepsize, in percent
        Valid values: 0 <= stepsize_jitter <= 1
        Defaults to 0
...
```

Following the hierarchy, the maximum tree depth derives from `nuts`, which itself is a value for the argument `engine` which derives from `hmc`. Adding this to our previous call we attempt

```
> ./model method=sample                          \
              algorithm=hmc                      \
                  metric=unit_e                  \
                  engine=nuts max_depth=-15      \
          data file=model.data.R                 \
          init=model.init.R                      \
          random seed=5                          \
```

which yields

```
-1 is not a valid value for "max_depth"
  Valid values: 0 < max_depth
Failed to parse arguments, terminating Stan
```

Where did that negative sign come from? Clumsy fingers are nothing to be embarrassed about, especially with such complex argument configurations. Removing the guilty character, we try

```
> ./model method=sample                    \
              algorithm=hmc                 \
                  metric=unit_e             \
                  engine=nuts max_depth=15  \
          data file=model.data.R            \
          init=model.init.R                 \
          random seed=5
```

which finally runs without issue.

9.6. Command Templates

This section provides templates for all of the arguments deriving from each of the possible methods: `sample`, `optimize`, `variational` and `diagnose`. Arguments in square brackets are optional, those not in square brackets are required for the template.

Sampling Templates

The No-U-Turn sampler (NUTS) is the default (and recommended) sampler for Stan. The full set of configuration options is in Figure 9.2.

A standard Hamiltonian Monte Carlo (HMC) sampler with user-specified integration time may also be used. Its set of configuration options are shown in Figure 9.3.

Both NUTS and HMC may be configured with either a unit, diagonal or dense Euclidean metric, with a diagonal metric the default.[3] A unit metric provides no parameter-by-parameter scaling, a diagonal metric scales each parameter independently, and a dense metric also rotates the parameters so that correlated parameters may move together. Although dense metrics offer the hope of superior simulation

[3] In Euclidean HMC, a diagonal metric emulates different step sizes for each parameter. Explicitly varying step sizes were used in Stan 1.3 and before; Neal (2011) discusses the equivalence.

```
> ./my_model sample                                          \
                 algorithm=hmc                                \
                     engine=nuts                              \
                       [max_depth=<int>]                      \
                     [metric={unit_e,diag_e,dense_e}]         \
                     [metric_file=<string>]                   \
                     [stepsize=<double>]                      \
                     [stepsize_jitter=<double>]               \
                 [num_samples=<int>]                          \
                 [num_warmup=<int>]                           \
                 [save_warmup=<boolean>]                      \
                 [thin=<int>]                                 \
                 [adapt                                       \
                      [engaged=<boolean>]                     \
                      [gamma=<double>]                        \
                      [delta=<double>]                        \
                      [kappa=<double>]                        \
                      [t0=<double>] ]                         \
             [data file=<string>]                             \
             [init=<string>]                                  \
             [random seed=<int>]                              \
             [output                                          \
                  [file=<string>]                             \
                  [diagnostic_file=<string>]                  \
                  [refresh=<int>] ]
```

Figure 9.2: *Command skeleton for invoking the no-U-turn sampler (NUTS). This is the same skeleton as that for basic HMC in Figure 9.3. Elements in braces are optional. All arguments and their default values are described in detail in Section 9.5.*

performance, they require more computation per iteration. Specifically for m samples of a model with n parameters, the dense metric requires $\mathcal{O}(n^3 \log(m) + n^2\, m)$ operations, whereas diagonal metrics require only $\mathcal{O}(n\, m)$. Furthermore, dense metrics are difficult to estimate, given the $\mathcal{O}(n^2)$ components with complex interdependence.

Optimization Templates

CmdStan supports several optimizers. These share many of their configuration options with the samplers. The default optimizer is the the limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method; Nocedal and Wright (2006) contains

```
> ./my_model sample                                          \
                 algorithm=hmc                                \
                     engine=static                            \
                       [int_time=<double>]                    \
                     [metric={unit_e,diag_e,dense_e}]         \
                     [metric_file=<string>]                   \
                     [stepsize=<double>]                      \
                     [stepsize_jitter=<double>]               \
                 [num_samples=<int>]                          \
                 [num_warmup=<int>]                           \
                 [save_warmup=<boolean>]                      \
                 [thin=<int>]                                 \
                 [adapt                                       \
                      [engaged=<boolean>]                     \
                      [gamma=<double>]                        \
                      [delta=<double>]                        \
                      [kappa=<double>]                        \
                      [t0=<double>] ]                         \
               [data file=<string>]                           \
               [init=<string>]                                \
               [random seed=<int>]                            \
               [output                                        \
                    [file=<string>]                           \
                    [diagnostic_file=<string>]                \
                    [refresh=<int>] ]
```

Figure 9.3: *Command skeleton for invoking the basic Hamiltonian Monte Carlo sampler (HMC). This is the same as the NUTS command skeleton shown in Figure 9.2 other than for the engine. Elements in braces are optional. All arguments and their default values are described in detail in Section 9.5.*

an excellent overview of both the BFGS and L-BFGS algorithms. The command skeleton for L-BFGS is in Figure 9.4 and the one for BFGS is in Figure 9.5. Stan also supports Newton's method; see (Nocedal and Wright, 2006) for more information. This method is the least efficient of the three, but has the advantage of setting its own step size. Other than not having a stepsize argument, the skeleton for Newton's method shown in Figure 9.6 is identical to that for BFGS.

```
> ./my_model optimize                                    \
                algorithm=lbfgs                           \
                    [init_alpha=<double>]                 \
                    [tol_obj=<double>]                    \
                    [tol_rel_obj=<double>]                \
                    [tol_grad=<double>]                   \
                    [tol_rel_grad=<double>]               \
                    [tol_param=<double>]                  \
                    [history_size=<int>]                  \
                [iter=<int>]                              \
                [save_iterations=<boolean>]               \
            [data file=<string>]                          \
            [init=<string>]                               \
            [random seed=<int>]                           \
            [output                                       \
                 [file=<string>]                          \
                 [diagnostic_file=<string>]               \
                 [refresh=<int>] ]
```

Figure 9.4: *Command skeleton for invoking the L-BFGS optimizer. All arguments and their default values are described in detail in Section 9.5.*

Variational Inference Templates

CmdStan implements Automatic Differentiation Variational Inference (Kucukelbir et al., 2015). The command skeleton for the `meanfield` algorithm is in Figure 9.7. The command skeleton for the `fullrank` algorithm is in Figure 9.8.

Diagnostic Command Skeleton

Stan reports on gradients for the model at a specified or randomly generated initial value. The command-skeleton in this case is very simple, and shown in Figure 9.9.

Generate_quantities Command Skeleton

The `generate_quantities` method can be used to generate additional quantities of interest from the model conditioned on the data. The data used to fit the model should be specified as part of the command.

```
> ./my_model optimize                                    \
                algorithm=bfgs                           \
                    [init_alpha=<double>]                \
                    [tol_obj=<double>]                   \
                    [tol_rel_obj=<double>]               \
                    [tol_grad=<double>]                  \
                    [tol_rel_grad=<double>]              \

                    [tol_param=<double>]                 \
                [iter=<int>]                             \
                [save_iterations=<boolean>]              \
            [data file=<string>]                         \
            [init=<string>]                              \
            [random seed=<int>]                          \
            [output                                      \
                 [file=<string>]                         \
                 [diagnostic_file=<string>]              \
                 [refresh=<int>] ]
```

Figure 9.5: *Command skeleton for invoking the BFGS optimizer. All arguments and their default values are described in detail in Section 9.5.*

```
> ./my_model optimize                                    \
                algorithm=newton                         \
                [iter=<int>]                             \
                [save_iterations=<boolean>]              \
            [data file=<string>]                         \
            [init=<string>]                              \
            [random seed=<int>]                          \
            [output                                      \
                 [file=<string>]                         \
                 [diagnostic_file=<string>]              \
                 [refresh=<int>] ]
```

Figure 9.6: *Command skeleton for invoking the Newton optimizer. All arguments and their default values are described in detail in Section 9.5.*

```
> ./my_model variational                                  \
                algorithm=meanfield                       \
                    [iter=<int>]                          \
                    [grad_samples=<int>]                  \
                    [elbo_samples=<int>]                  \
                    [eta=<double>]                        \
                    [adapt                                \
                     [engaged=<boolean>]                  \
                     [iter=<int>] ]                       \
                    [tol_rel_obj=<double>]                \
                    [eval_elbo=<int>]                     \
                    [output_samples=<int>]                \
              [data file=<string>]                        \
              [init=<string>]                             \
              [random seed=<int>]                         \
              [output                                     \
                   [file=<string>]                        \
                   [diagnostic_file=<string>]             \
                   [refresh=<int>] ]
```

Figure 9.7: *Command skeleton for invoking the meanfield variational inference algorithm. All arguments and their default values are described in detail in Section 9.5.*

```
> ./my_model variational                                  \
                algorithm=fullrank                        \
                    [iter=<int>]                          \
                    [grad_samples=<int>]                  \
                    [elbo_samples=<int>]                  \
                    [eta=<double>]                        \
                    [adapt                                \
                     [engaged=<boolean>]                  \
                     [iter=<int>] ]                       \
                    [tol_rel_obj=<double>]                \
                    [eval_elbo=<int>]                     \
                    [output_samples=<int>]                \
              [data file=<string>]                        \
              [init=<string>]                             \
              [random seed=<int>]                         \
              [output                                     \
                   [file=<string>]                        \
                   [diagnostic_file=<string>]             \
                   [refresh=<int>] ]
```

Figure 9.8: *Command skeleton for invoking the fullrank variational inference algorithm. All arguments and their default values are described in detail in Section 9.5.*

```
> ./my_model diagnose                  \
                [test=gradient]        \
                    [epsilon=<real>]   \
                    [error=<real>]     \
              [data file=<string>]     \
              [init=<string>]          \
              [random seed=<int>]      \
```

Figure 9.9: *Command skeleton for invoking model diagnostics. All arguments and their default values are described in detail in Section 9.5.*

```
> ./my_model generate\_quantities                    \
                [fitted\_params=<string>]        \
             [data file=<string>]       \
             [random seed=<int>]        \
```

Figure 9.10: *Command skeleton for generating new quantities of interest from a set of fitted parameter values using* `method=generate_quantities`. *All arguments and their default values are described in detail in Section 9.5.*

Appendices

A. Licensing

CmdStan, Stan, and Stan's four dependent libraries, Stan Math Library, Boost, Eigen, Sundials, and Intel TBB are distributed under liberal freedom-respecting[1] licenses approved by the Open Source Initiative.[2]

In particular, the licenses for CmdStan and its dependent libraries have no "copy-left" provisions requiring applications of CmdStan to be open source if they are redistributed.

However, the Apache 2.0 license of the Intel TBB is not compatible with the GPL-2 (while it is compatible with GPL-3). This in-compatibility implies that no unitary binary of Apache 2.0 work (part of CmdStan) and GPL-2 work can be distributed. A detailed discussion on the Apache 2.0 license implications for Stan can be found on our wiki https://github.com/stan-dev/math/wiki/Apache-2.0-License-Evaluation.

This chapter describes the licenses for the tools that are distributed with CmdStan. The next chapter explains some of the build tools that are not distributed with CmdStan, but are required to build and run Stan models.

A.1. CmdStan's License

CmdStan is distributed under the BSD 3-clause license (BSD New).

http://www.opensource.org/licenses/BSD-3-Clause

A.2. Stan's License

Stan is distributed under the BSD 3-clause license (BSD New).

http://www.opensource.org/licenses/BSD-3-Clause

A.3. Stan Math Library's License

The Stan Math Library is distributed under the BSD 3-clause license (BSD New).

http://www.opensource.org/licenses/BSD-3-Clause

[1] The link http://www.gnu.org/philosophy/open-source-misses-the-point.html leads to a discussion about terms "open source" and "freedom respecting."

[2] See http://opensource.org.

A.4. Boost License

Boost is distributed under the Boost Software License version 1.0.

http://www.opensource.org/licenses/BSL-1.0

A.5. Eigen License

Eigen is distributed under the Mozilla Public License, version 2.

http:/http://opensource.org/licenses/mpl-2.0

A.6. Sundials License

Sundials is distributed under the BSD 3-clause license (BSD New).

http://computation.llnl.gov/projects/
sundials-suite-nonlinear-differential-algebraic-equation-solvers/
license

A.7. Intel Threading Building Blocks License

The Intel TBB is distributed under the Apache 2.0 license.

https://github.com/intel/tbb/blob/tbb_2019/LICENSE

A.8. Google Test License

CmdStan uses Google Test for unit testing; it is not required to compile or execute models. Google Test is distributed under the BSD 2-clause license.

http://www.opensource.org/licenses/BSD-License

B. Installation and Compatibility

This appendix describes the hardware and software required to run CmdStan. The software required includes CmdStan and its libraries, as well as a contemporary C++ compiler. CmdStan requires hardware powerful enough to build and execute the models. Ideally, that will be a 64-bit computer with at least 4GB of memory and multiple processor cores.

B.1. Operating System

CmdStan is written in portable C++ with C++11 and C++14 features, as are the libraries on which it depends. Therefore, CmdStan should run on any machine for which a suitable C++ compiler supporting C++1y or C++14 features is available. In practice, CmdStan, like the Boost and Eigen libraries on which it depends, is very hard on the compiler and linker.

CmdStan has been tested on the following operating systems.

- Linux (Debian, Ubuntu)
- Mac OS X (from 10.6 "Snow Leopard" through 10.14 "Mojave")
- Windows (7, 8, 10).

CmdStan should work on other versions of these operating systems if compatible C++ compilers can be found. The plan is to keep up with new versions of these operating systems and gradually phase out testing on older versions.

B.2. Required Software and Tools

The only two absolute requirements for running CmdStan are the CmdStan source code (and dependent libraries) and a C++ compiler.

CmdStan Source

In order to compile Stan program, the CmdStan source code is required. The CmdStan source code distribution includes CmdStan's source code, Stan's source code, documentation, build scripts, unit tests, documentation and source for the required libraries, Stan Math Library, Boost, Eigen, Sundials, Intel TBB, and the source for an optional testing library, Google Test.

CmdStan: Stable Releases

The latest release version of CmdStan can be downloaded from the CmdStan home page:

> http://mc-stan.org/cmdstan.html

CmdStan: Development Source Control

The source code repository is hosted by GitHub, and contains the latest versions of CmdStan (and Stan) underdevelopment. See:

> http://mc-stan.org/source-repos.html

Stan Library Source

The source code for Stan's parse and implementation of inference algorithms are in the Stan library.

- Home: http://mc-stan.org/stan.html
- License: BSD
- Tested Version: 2.21.0

The Stan source code is distributed with CmdStan.

Stan Math Library Source

Stan's mathematical functions and inference algorithm rely on the reverse-mode automatic differentiation implemented in the Stan Math Library.

- Home: http://mc-stan.org
- License: BSD
- Tested Version: 2.21.0

The Stan Math Library source code is distributed with CmdStan

Boost C++ Library Source

Stan's parser and some of its mathematical functions and template metaprogramming facilities are implemented with the Boost C++ Library.

- Home: http://www.boost.org/users/license.html
- License: Boost Software License
- Tested Version: 1.69.0

The Boost source code is distributed with CmdStan.

Eigen Matrix and Linear Algebra Library Source

Stan's matrix algebra depends on the Eigen C++ matrix and linear algebra library.

- Home: http://eigen.tuxfamily.org
- License: Mozilla Public License, version 2.0
- Tested Version: 3.3.3

The Eigen source code is distributed with CmdStan.

SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers

Some of Stan's ordinary differenetial equation (ODE) solvers and implicit function solvers use the Sundials library.

- Home: https://computing.llnl.gov/projects/sundials
- License: BSD
- Tested Version: 4.1.0

The Sundials source code is distributed with CmdStan.

Intel Threading Building Blocks

Stan uses the Intel TBB library for advanced threading support.

- Home: https://software.intel.com/en-us/tbb
- License: Apache 2.0
- Tested Version: 2019 update 8

The Intel TBB source code is distributed with CmdStan.

C++ Compiler

Compiling CmdStan programs requires a C++ compiler. CmdStan has been primarily developed with `clang++` and `g++` and no promises are made for other compilers. The full set of compilers for which CmdStan has been tested is listed here: https://github.com/stan-dev/stan/wiki/Supported-C---Compilers-and-Language-Features

B.3. Step-by-Step Windows Install Instructions

CmdStan has been tested on Windows XP, Windows 7, and Windows 8.

CmdStan also runs under Cygwin, which provides a unix-like shell on top of Windows. Instructions for Cygwin installation are provided below in their own subsection.

Windows Tips

Opening a Command Shell

To open a Windows command shell, first open the `Start Menu` (usually in the lower left of the screen), select option `All Programs`, then option `Accessories`, then program `Command Prompt`.

Alternatively, enter `[Windows+r]` (both keys together on the keyboard), and enter `cmd` into the text field that pops up in the Run window, then press `[Return]` on the keyboard to run.

32-bit Builds

CmdStan defaults to a 64-bit build. On a 32-bit operating system, include `BIT=32` in `make/local`. See Appendix B.6 for more details.

Rtools C++ Development Environment

The simplest way to install a full C++ build environment that will work for CmdStan is to use the Rtools package designed for R developers on Windows (even if you don't plan to use R).

First, download the latest *frozen* (i.e., stable) version of Rtools from the Rtools home page, using

http://cran.r-project.org/bin/windows/Rtools/

Next, double click on the downloaded file to open the Rtools install wizard, then proceed through its options.

- *Language*: select language, click `Next`,
- *Welcome*: click `Next`,
- *Information*: click `Next`,
- *Setup Location*: accept default (`c:\Rtools`), click `Next`,
- *Select Components*: select default, `Package Authoring`, click `Next`,
- *Select Additional Tasks*: check `Edit Path` and `Save Version in Registry`, click `Next`,
- *System Path Report*: ensure that that the paths to `c:\Rtools\bin` and `c:\Rtools\gcc-4.6.3\bin` are listed at the beginning of the path and click `Next`,
- *Ready to Install*: click `Next`, wait for the install to complete, then
- *Finish*: click `Finish`.
- *Confirm Path*: After the install has completed, open a command prompt and type `PATH` to ensure that the new path is activated and the Rtools folders are in the system path.

Checking the Path

Make sure that `c:\Rtools\bin` has been added to your `PATH` environment variable, and then open another command window. You should be able to follow the last step, *Comfirm Path*, above.

Note: if you see an error like "`FIND: Parameter format not correct`," please check to see that `c:\Rtools\bin` is listed at the beginning of the path.

Verifying Tools

To verify that g++ is installed, use the following command.

```
> g++ -v
```

This should report version information for g++. Next, verify that make variant `mingw32-make` is installed with the following command.

```
> mingw32-make -v
```

This should print version information for make.

Downloading and Unpacking CmdStan

The CmdStan source code distribution is named `cmdstan-2.21.0.tar.gz`; the versions here are major version 2, minor version 7, and patch level 0. Download the latest CmdStan source tarball from the CmdStan downloads page,

> https://github.com/stan-dev/cmdstan/releases

to any non-temporary folder. (If in doubt, select `My Documents` on Windows XP or `Documents` on Windows 7.)

Change to the download directory (aka folder) using one of the following commands, replacing `<username>` with a Windows user name.

- *Windows XP*: From the default starting directory, use the following commands (quotes and all):

  ```
  > cd "My Documents"
  ```

  The full path (including quotes) will work from anywhere,

  ```
  > cd "c:\Documents and Settings\<username>\My Documents"
  ```

- *Windows 7*: From the default starting directory, use

  ```
  > cd Documents
  ```

  or use the full path, including quotes, from anywhere,

  ```
  > cd "c:\Users\<username>\Documents"
  ```

To verify that the downloaded CmdStan `.tar.gz` file is there, list the directory contents using:

```
> dir
```

Finally, unpack the distribution using the `tar` command (which is installed as part of Rtools).

```
> tar --no-same-owner -xzf cmdstan-2.21.0.tar.gz
```

The `-no-same-owner` flag is not strictly necessary, but it removes a bunch of irrelevant warnings.

64-bit Cygwin Install Instructions

CmdStan can be run under Cygwin, the Unix look-and-feel environment for Windows. Cygwin must have recent versions of `make` and `g++` (part of gcc) installed. Within a Cygwin shell, CmdStan will behave as under other Unixes. Follow the directions in Appendix B.5.

B.4. Step-by-Step Mac Install Instructions

This section provides step-by-step install instructions for the Mac. CmdStan has been tested on Mac OS X versions Mavericks, Snow Leopard, Lion, and Mountain Lion.

Install Xcode C++ Development Environment

The easiest (but not the only) way to install a C++ development environment on a Mac is to use Apple's Xcode development environment.

From the Xcode home page,

> https://developer.apple.com/xcode/

click `View in Mac App Store`.

From the App Store, click `Install`, enter an Apple ID, and wait for Xcode to finish installing.

Open the Xcode application, click top-level menu `Preferences`, click top-row button `Downloads`, click button for `Components`, click on the `Install` button to the right of the `Command Line Tools` entry, then wait for it to finish installing.

Click the top-level menu item `Xcode`, then click item `Quit Xcode` to quit.

To test, open the Terminal application and enter

```
> make --version
> g++ --version
```

Verify that `make` is at version 3.81 or later and `g++` is at 4.9.3 or later.

Download and Unpack CmdStan Source

Download the most recent version of `cmdstan-2.21.0.tar.gz` from the CmdStan downloads list,

> https://github.com/stan-dev/cmdstan/releases

Open the folder containing the download in the Finder (typically, the user's top-level `Downloads` folder).

If the Mac OS has not automatically unpacked the `.tar.gz` file into file `cmdstan-2.21.0.tar`, double-click the `.tar.gz` file to unpack.

Double click on the `.tar` file to unarchive directory `cmdstan-2.21.0`.

Move the resulting directory to a location where it will not be deleted, henceforth called `<cmdstan-home>`.

B.5. Step-by-Step Linux Install Instructions

CmdStan has been tested on various Linux installations, including Ubuntu, Debian, and Red Hat.

Installing C++ Development Tools

On Linux, C++ compilers and `make` are often installed by default.

To see if the g++ compiler and `make` build system are already installed, use the commands

```
> g++ --version
> make --version
```

If these are at least at g++ version 4.9.3 or later and `make` version 3.81 or later, no additional installations are necessary. It may still be desirable to update the C++ compiler g++, because later versions are faster.

To install the latest version of these tools (or upgrade an older version), use the following commands or their equivalent for your distribution:

```
> sudo apt install g++
> sudo apt install make
```

A password will likely be required by the superuser command `sudo`.

Downloading and Unpacking CmdStan Source

Download the most recent stable version of CmdStan, `cmdstan-2.21.0.tar.gz`, from the CmdStan downloads page,

https://github.com/stan-dev/cmdstan/releases

to the directory where Stan will reside.

In a command shell, change directories to where the tarball was downloaded, say `<download-dir>`, with

```
> cd <download-dir>
```

where <download-dir> is replaced with the actual path to the directory.

Then, unpack the distribution into the subdirectory

```
<download-dir>/cmdstan-2.21.0
```

with

```
> tar -xzf cmdstan-2.21.0.tar.gz
```

B.6. make Options

Setting a Variable

CmdStan uses make for building the CmdStan tools and compiling Stan programs as executables. Users can customize how the tools are built by creating and editing make/local inside their <cmdstan-home> directory.

Customization is done by setting variables or appending to them. For each variable that needs to be set, write <variable>=<value> on its own line inside make/local. To append to existing variables, write <variable>+=<value>.

Customizing make Options

Compiler Settings

The defaults should work for most Windows, Linux, and Mac setups. The most common options to customize the build are:

- CXX. The compiler used to build CmdStan and the Stan executables. The default is C++ compiler on the system.
- O. The optimization level for the compiler. The default is O=3. Both g++ and clang++ recognize 0,1,2,3, and s.
- O_STANC. The optimization level for building the Stan compiler. The default is O=3 on Windows and O=0 for other operating systems.
- CXXFLAGS_OS. Any additional compiler flags that would be useful to set.
- INC_FIRST. Any header files to include before the rest of the Stan includes.

`stanc` *Options*

There are two options for controlling `stanc`:

- `STANCFLAGS`. These flags are passed to `stanc`. To allow functions in Stan programs that are declared, but undefined, use `STANCFLAGS = --allow_undefined` and update `USER_HEADER` appropriately.

- `USER_HEADER`. If `--allow_undefined` is passed to `stanc`, the value of this variable is the additional include that is passed to the C++ linker when compiling the program. The file in `USER_HEADER` must have a definition of the function if the function is used and not defined in the Stan program. This defaults to `user_header.hpp` in the directory of the Stan program being compiled.

Changing Library Locations

The next set of variables allow for easy replacement of dependent libraries. CmdStan depends on Stan, Boost, Eigen, Sundials, and Intel TBB. Although CmdStan is bundled with a particular verison of Stan, other versions can be used. The same is true with Boost, Eigen, Sundials, and Intel TBB.

The Stan developers typically replace the tagged version of Stan in `<cmdstan-home>/stan` with an updated version and do not set these variables. That said, it is easy to point CmdStan to other versions of these libraries in other directories.

- `STAN`. The location of the Stan source directory. The default is `STAN=stan_2.21.0/`. Note: the trailing forward slash is necessary.

- `MATH`. The location of the Stan Math Library. The default is `MATH=stan/l../stan_math_2.21.0/`.

- `BOOST`. The location of the Boost library. If `BOOST` is not explicitly set, this will default to the `lib/boost_1.69.0/` folder within `MATH`.

- `EIGEN`. The location of the Eigen library. If `EIGEN` is not explicitly set, this will default to the `lib/eigen_3.3.3/` folder within `MATH`.

- `SUNDIALS`. The location of the SUNDIALS library. If `SUNDIALS` is not explicity set, this will default to the `lib/sundials_4.1.0/` folder within `MATH`.

- `TBB`. The location of the Intel TBB library. If `TBB` is not explicity set, this will default to the `lib/tbb_2019_U8` folder within `MATH`.

Rebuilding CmdStan

When compiler flags are changed, CmdStan needs to be rebuilt in order for the changes to take place. The easiest way to do this is to type:

```
> make clean-all
```

Then rebuild CmdStan and the Stan programs of interest.

B.7. MKL Compiler Instructions

Getting the MKL

To purchase a license, see

http://software.intel.com/en-us/intel-mkl

For non-commercial development, see

http://software.intel.com/non-commercial-software-development

Compiling with MKL

In order to use Intel's math kernel library (MKL) for C++, a few lines need to be added to `make/local`.

- `CXX = icc`. Set the compiler to icc.
- `MKLROOT = /apps/intel/2013/mkl`. Create a new variable with the location of the MKL path.
- `CXXFLAGS += -I $(MKLROOT)/include -DEIGEN_USE_MKL_ALL`. This tells Eigen to use MKL and where to find the necessary header files.
- `LDLIBS += -L$(MKLROOT)/lib/intel64 -lmkl_intel_lp64` and `LDLIBS += -lmkl_core -lmkl_sequential -lpthread -lm`. This links in the MKL library for the CmdStan executables. The exact implementation will depend on the particular system. Use the MKL link line advisor for help.

Note: Make sure to do the above changes before compiling for the first time - otherwise Stan will be compiled with g++ and you won't see any performance gains.

Additional compiler options

By default, Intel's compiler trades off accuracy for speed. This, unfortunately, isn't ideal behavior for the inference algorithms and may cause divergent transitions to go *unreported*. Add these flags to `make/local` to force more accurate floating point computations which ensure that divergent transitions are reported:

- `CXXFLAGS += -fp-model precise -fp-model source`

B.8. Optional Parallelization Support

CmdStan has optional support for within-chain parallelization using threading, OpenCL, or MPI (Message Passing Interface).

Threading and MPI

Threads can utilize multiple cores on a single machine. Please refer to https://github.com/stan-dev/math/wiki/Threading-Support for instructions on how to setup threading.

MPI is typically used on large computing clusters. The MPI standard allows to link together a large number of processes which can run locally on one machine or across many different machines. Please refer to https://github.com/stan-dev/math/wiki/MPI-Parallelism for more details.

The within-chain parallelization for threading and MPI is used in the Stan language to parallelize the evaluation of the `map_rect` command. Note that MPI parallelism will take precendence in case both features are enabled.

OpenCL

OpenCL in CmdStan enables Cholesky decompositions to utilize a GPU for large parallel computations. In order to use OpenCL, a few lines need to be added to `make/local`.

- `STAN_OPENCL=true` Enable the use of OpenCL.
- `OPENCL_PLATFORM_ID=<int>` Set which OpenCL platform to use.
- `OPENCL_DEVICE_ID=<int>` Set which OpenCL device on the selected platform to use.

Windows users need to additionally specify the following two flags depending on the GPU you are running OpenCL on.

NVIDIA GPU

- `LDFLAGS_OPENCL= -L"\$(CUDA_PATH)\lib\x64" -lOpenCL`

AMD GPU

- `LDFLAGS_OPENCL= -L"\$(AMDAPPSDKROOT)lib\x86_64" -lOpenCL`

If you are unsure if your devices support OpenCL, we suggest running the `clinfo` application that lists all available OpenCL supported devices. Please refer to https://github.com/stan-dev/math/wiki/OpenCL-GPU-Routines for additional setup instructions. Currently, OpenCL will only pass matrices over to the GPU if their dimensions are larger than 1250×1250.

B.9. Optional Components for Developers

CmdStan is developed using the following set of tools. The various command examples in this manual have assumed they can be found on the command path. The makefile allows precise locations to be plugged in.

GNU Make Build Tool

CmdStan automates the build, test, documentation, and deployment tasks using scripts in the form of makefiles to run with GNU Make.

- Home: http://www.gnu.org/software/make
- License: GPLv3+
- Tested Versions: 3.81 (Mac OS X), 3.79 (Windows 7)

On Windows `mingw32-make` variant must be used as a requirement to build the Intel TBB. This variant of `make` is available as part of RTools https://cran.rstudio.com/bin/windows/Rtools/.

Doxygen Documentation Generator

CmdStan's API documentation is generated using the Doxygen Tool.

- Home: http://www.stack.nl/~dimitri/doxygen/index.html
- License: GPL2
- Tested Version(s): Mac OS X 1.8.2, Windows 1.8.2

Git Version Control System

CmdStan uses the Git version control system for its software, libraries, and documentations. Git is required to interact with the most recent versions of code in the version control repository.

- Home: http://git-scm.com/
- License: GPL2
- Tested Version(s): Mac versions 1.8.2.3 and 1.7.8.4; Windows version 1.7.9

Google Test C++ Testing Framework

CmdStan's unit testing is based on the Google's googletest C++ testing framework.

- Home: http://code.google.com/p/googletest/
- License: BSD
- Tested Version(s): 1.8.1

The Google Test framework is distributed with Stan.

B.10. Tips for Mac OS X

Finding and Opening Mac Applications and Files

To open an application, use `[Command-Space]` (press both keys at once on the keyboard) to open Spotlight, enter the application's name in the text field, then click on the application in the pop-up menu or `[Return]` if the right file or application is highlighted.

Spotlight can be used in the same way to find files or folders, such as the default `Downloads` folder for web downloads.

Open a Terminal for Shell Commands

To run shell commands, open the built-in Terminal application (see the previous subsection for details on how to find and open applications).

Install Xcode

Apple's Xcode contains both the `clang++` and `g++` compilers and make, all of the tools needed to work with CmdStan as a user. The version of Xcode to install depends on the version of Mac OS X.

Alternative, GCC-Only Installer

A stripped down installer for just the GCC package, including the C++ compilers g++ and clang++, available for Mac OS X 10.6 ("Snow Leopard") or later,

https://github.com/kennethreitz/osx-gcc-installer/

The fill list of tools in this distribution is available at:

http://www.opensource.apple.com/release/developer-tools-41/

More Recent Compilers

Alternative compilers to those distributed by Apple as part of Xcode are available at the following locations.

Homebrew

One way to get pre-built binaries for Mac OS X is to use Homebrew, which is available from the following link.

http://mxcl.github.com/homebrew/

MacPorts

MacPorts hosts recent versions of compilers for the Macintosh.

https://distfiles.macports.org/MacPorts/

After finding the appropriate `.dmg` file, clicking on it, then double clicking on the resulting `.pkg` file, and clicking through some more menus, the following will need to be entered from a terminal window to install it.

```
> sudo port install gccVersion
```

In this command, *gccVersion* is the name of a compiler version, such as `g++-mp-4.6`, for version 4.6. Errors may arise during the install such as the following.

```
Error: Target org.macports.activate returned: Image error:
/opt/local/include/gmp.h already exists and does not belong to
a registered port. Unable to activate port gmp. Use 'port -f
activate gmp' to force the activation.
```

This issue can be resolved by running the following command.

```
> sudo port -f activate gmp
```

LaTeX Typesetting Package

CmdStan uses the LaTeX typesetting package for generating manuals, talks, and other materials (Doxygen is used for API documentation; see below). The first step is to download the MacTeX `.mpkg` file from the following URL [warning: the download is approximately 2GB and the installation approximately 3.5GB].

> http://www.tug.org/mactex/2011/

Once it is downloaded, just click on the `.mpkg` file and then follow the installer instructions. The installer will add the command to the `PATH` environment variable so that the `pdflatex` used by Stan is available from the command line.

C. JSON Format

CmdStan can use JSON format for input data for both model data and parameters. Model data is read in by the model constructor. Model parameters are used to initialize the sampler and optimizer.

C.1. JSON Syntax Summary

JSON is a data interchange notation, defined by an ECMA standard. JSON data files must in Unicode. JSON data is a series of structural tokens, literal tokens, and values:

- Structural tokens are the left and right curly bracket, left and right square bracket, the semicolon, and the comma. `{}[]:,`
- Literal tokens must always be in lowercase. There are three literal tokens: `true` `false` `null`
- A primitive value is a single token which is either a literal, a string, or a number.
- A string consists of zero or more Unicode characters enclosed in double quotes, e.g. `"foo"`. A backslash is used to escape the double quote character as well as the backslash itself. JSON allows the use of Unicode character escapes, e.g.
  `uHHHH` where `HHHH` is the Unicode code point in hex.
- All numbers are decimal numbers. Scientific notation is allowed. The following are examples of numbers `17 17.2 -17.2 -17.2e8 17.2e-8` The concepts of positive and negative infinity as well as "not a number" cannot be expressed as numbers in JSON, but they can be encoded as strings which can be mixed with numbers.
- A JSON array is an ordered, comma-separated list of zero or more JSON values enclosed in square brackets. The elements of an array can be of any type. The following are examples of arrays: `[] [1] ["a","b",true]`
- A name-value pair consists of a string followed by a colon followed by a value, either primitive or compound.
- A JSON object is a comma-separated series of zero or more name-value pairs enclosed in curly brackets. Each name-value pair is a member of the object.

Membership is unordered. Member names are not required to be unique. The following are examples of objects: `{} {"foo": null} {"bar" : 17, "baz" : [14,15,16.6] }`

C.2. Stan Data Types in JSON Notation

Stan follows the JSON standard. A Stan input file in JSON notation consists of JSON object which contains zero or more name-value pairs. This structure corresponds to a Python data dictionary object. The following is an example of JSON data for the simple Bernoulli example model:

```
{ "N" : 10, "y" : [0,1,0,0,0,0,0,0,0,1] }
```

Matrix data and multi-dimensional arrays are indexed in row-major order. For a Stan program which has data block

```
data {
  int d1;
  int d2;
  int d3;
  int ar[d1, d2, d3];
}
```

the following JSON input file would be valid

```
{ "d1" : 2,
  "d2" : 3,
  "d3" : 4,
  "ar" : [[[0,1,2,3], [4,5,6,7], [8,9,10,11]],
          [[12,13,14,15], [16,17,18,19], [20,21,22,23]]]
}
```

JSON ignores whitespace. In the above examples, the spaces and newlines are only used to improve readability and can be omitted.

All data inputs are encoded as name-value pairs. The following table provides more examples of JSON data. The left column contains a Stan data variable declaration and the right column contains valid JSON data inputs.

Stan variable	*JSON data*
`int i;`	`"i" : 17`

`real a;`	`"a" : 17`
	`"a" : 17.2`
	`"a" : "NaN"`
	`"a" : "+inf"`
	`"a" : "-inf"`

`int a[5];`	`"a" : [1, 2, 3, 4, 5]`
`real a[5];`	`"a" : [ 1, 2, 3.3, "NaN", 5 ]`
`vector[5] a;`	`"a" : [ 1, 2, 3.3, "NaN", 5 ]`
`row_vector[5] a;`	`"a" : [ 1, 2, 3.3, "NaN", 5 ]`
`real a[5];`	`"a" : [ 1, 2, 3.3, "NaN", 5 ]`
`matrix[2,3] a;`	`"a" : [ [ 1, 2, 3 ], [ 4, 5, 6] ]`

D. Dump Data Format

For representing structured data in files, CmdStan uses the dump format introduced in S and used in R and JAGS (and in BUGS, but with a different ordering). A dump file is structured as a sequence of variable definitions. Each variable is defined in terms of its dimensionality and its values. There are three kinds of variable declarations, one for scalars, one for sequences, and one for general arrays.

D.1. Creating Dump Files

Dump files can be created from R using RStan. The function is `stan_rdump` in package `rstan`.

Using R's native `dump()` function can produce dump files which Stan cannot read in. The underlying cause is that R supports complicated data structures, some of which are not used in CmdStan. For example, R's `dump()` can write a numerical vector with names for each element.

D.2. Scalar Variables

A simple scalar value can be thought of as having an empty list of dimensions. Its declaration in the dump format follows the S assignment syntax. For example, the following would constitute a valid dump file defining a single scalar variable `y` with value 17.2.

```
y <- 17.2
```

A scalar value is just a zero-dimensional array value.

D.3. Sequence Variables

One-dimensional arrays may be specified directly using the S sequence notation. The following example defines an integer-value and a real-valued sequence.

```
n <- c(1,2,3)
y <- c(2.0,3.0,9.7)
```

Arrays are provided without a declaration of dimensionality because the reader just counts the number of entries to determine the size of the array.

Sequence variables may alternatively be represented with R's colon-based notation. For instance, the first example above could equivalently be written as

```
n <- 1:3
```

The sequence denoted by `1:3` is of length 3, running from 1 to 3 inclusive. The colon notation allows sequences going from high to low, as in the first of the following examples, which is equivalent to the second.

```
n <- 2:-2
n <- c(2,1,0,-1,-2)
```

As a special case, a sequence of zeros can also be represented in the dump format by `integer(x)` and `double(x)`, for type int and double, respectively. Here `x` is a non-negative integer to specify the length. If `x` is `0`, it can be ommitted. The following are some examples.

```
x1 <- integer()
x2 <- integer(0)
x3 <- integer(2)
y1 <- double()
y2 <- double(0)
y3 <- double(2)
```

D.4. Array Variables

For more than one dimension, the dump format uses a dimensionality specification. For example,

```
y <- structure(c(1,2,3,4,5,6), .Dim = c(2,3))
```

This defines a 2×3 array. Data is stored in column-major order, meaning the values for `y` will be as follows.

```
y[1,1] = 1     y[1,2] = 3     y[1,3] = 5
y[2,1] = 2     y[2,2] = 4     y[2,3] = 6
```

The `structure` keyword just wraps a sequence of values and a dimensionality declaration, which is itself just a sequence of non-negative integer values. The product of the dimensions must equal the length of the array.

If the values happen to form a contiguous sequence of integers, they may be written with colon notation. Thus the example above is equivalent to the following.

```
y <- structure(1:6, .Dim = c(2,3))
```

The same applies to the specification of dimensions, though it is perhaps less likely to be used. In the above example, c(2,3) could be written as `2:3`.

Arrays of more than two dimensions are written in a last-index major form. For example,

```
z <- structure(1:24, .Dim = c(2,3,4))
```

produces a three-dimensional `int` (assignable to `real`) array `z` with values

```
z[1,1,1] =  1   z[1,2,1] =  3   z[1,3,1] =  5
z[2,1,1] =  2   z[2,2,1] =  4   z[2,3,1] =  6

z[1,1,2] =  7   z[1,2,2] =  9   z[1,3,2] = 11
z[2,1,2] =  8   z[2,2,2] = 10   z[2,3,2] = 12

z[1,1,3] = 13   z[1,2,3] = 15   z[1,3,3] = 17
z[2,1,3] = 14   z[2,2,3] = 16   z[2,3,3] = 18

z[1,1,4] = 19   z[1,2,4] = 21   z[1,3,4] = 23
z[2,1,4] = 20   z[2,2,4] = 22   z[2,3,4] = 24
```

The sequence of values inside `structure` can also be `integer(x)` or `double(x)`. In particular, if one or more dimensions is zero, `integer()` can be put inside `structure`. For instance, the following example is supported by the dump format.

```
y <- structure(integer(), .Dim = c(2, 0))
```

D.5. Matrix- and Vector-Valued Variables

The dump format for matrices and vectors, including arrays of matrices and vectors, is the same as that for arrays of the same shape.

Vector Dump Format

The following three declarations have the same dump format for their data.

```
real a[K];
vector[K] b;
row_vector[K] c;
```

Matrix Dump Format

The following declarations have the same dump format.

```
real a[M,N];
matrix[M,N] b;
```

Arrays of Vectors and Matrices

The key to undertanding arrays is that the array indexing comes before any of the container indexing. That is, an array of vectors is just that — provide an index and get a vector. See the chapter on array and matrix types in the user's guide section of the languag emanual for more information.

For the dump data format, the following declarations have the same arrangement.

```
real a[M,N];
matrix[M,N] b;
vector[N] c[M];
row_vector[N] d[M];
```

Similarly, the following also have the same dump format.

```
real a[P,M,N];
matrix[M,N] b[P];
vector[N] c[P,M];
row_vector[N] d[P,M];
```

D.6. Integer- and Real-Valued Variables

There is no declaration in a dump file that distinguishes integer versus continuous values. If a value in a dump file's definition of a variable contains a decimal point (e.g., `132.3`) or uses scientific notation (e.g., `1.323e2`), Stan assumes that the values are real.

For a single value, if there is no decimal point, it may be assigned to an `int` or `real` variable in Stan. An array value may only be assigned to an `int` array if there is no decimal point or scientific notation in any of the values. This convention is compatible with the way R writes data.

The following dump file declares an integer value for `y`.

```
y <- 2
```

This definition can be used for a Stan variable `y` declared as `real` or as `int`. Assigning an integer value to a real variable automatically promotes the integer value to a real value.

Integer values may optionally be followed by `L` or `l`, denoting long integer values. The following example, where the type is explicit, is equivalent to the above.

```
y <- 2L
```

The following dump file provides a real value for `y`.

```
y <- 2.0
```

Even though this is a round value, the occurrence of the decimal point in the value, `2.0`, causes Stan to infer that `y` is real valued. This dump file may only be used for variables `y` declared as real in Stan.

Scientific Notation

Numbers written in scientific notation may only be used for real values in Stan. R will write out the integer one million as `1e+06`.

Infinite and Not-a-Number Values

Stan's reader supports infinite and not-a-number values for scalar quantities (see the section of the reference manual section of the language manaul for more information on Stan's numerical data types). Both infinite and not-a-number values are supported by Stan's dump-format readers.

Value	*Preferred Form*	*Alternative Forms*
positive infinity	`Inf`	`Infinity`, `infinity`
negative infinity	`-Inf`	`-Infinity`, `-infinity`
not a number	`NaN`	

These strings are not case sensitive, so `inf` may also be used for positive infinity, or `NAN` for not-a-number.

D.7. Quoted Variable Names

In order to support JAGS data files, variables may be double quoted. For instance, the following definition is legal in a dump file.

```
"y" <- c(1,2,3)
```

D.8. Line Breaks

The line breaks in a dump file are required to be consistent with the way R reads in data. Both of the following declarations are legal.

```
y <- 2
y <-
3
```

Also following R, breaking before the assignment arrow are not allowed, so the following is invalid.

```
y
<- 2  # Syntax Error
```

Lines may also be broken in the middle of sequences declared using the `c(...)` notation., as well as between the comma following a sequence definition and the dimensionality declaration. For example, the following declaration of a $2 \times 2 \times 3$ array is valid.

```
y <-
structure(c(1,2,3,
4,5,6,7,8,9,10,11,
12), .Dim = c(2,2,
3))
```

Because there are no decimal points in the values, the resulting dump file may be used for three-dimensional array variables declared as `int` or `real`.

D.9. BNF Grammar for Dump Data

A more precise definition of the dump data format is provided by the following (mildly templated) Backus-Naur form grammar.

```
definitions ::= definition+

definition ::= name <- value optional_semicolon

name ::= char*
       | ''' char* '''
       | '"' char* '"'
```

```
value ::= value<int> | value<double>

value<T> ::= T
           | seq<T>
           | zero_array<T>
           | 'structure' '(' seq<T> ',' ".Dim" '=' seq<int> ')'
           | 'structure' '(' zero_array<T> ',' ".Dim" '=' seq<int> ')'

seq<int> ::= int ':' int
           | cseq<int>

zero_array<int> ::= "integer" '(' <non-negative int>? ')'

zero_array<real> ::= "double" '(' <non-negative int>? ')'

seq<real> ::= cseq<real>

cseq<T> ::= 'c' '(' vseq<T> ')'

vseq<T> ::= T
          | T ',' vseq<T>
```

The template parameters `T` will be set to either `int` or `real`. Because Stan allows promotion of integer values to real values, an integer sequence specification in the dump data format may be assigned to either an integer- or real-based variable in Stan.

Bibliography

Betancourt, M. (2012). A general metric for Riemannian manifold Hamiltonian Monte Carlo. *arXiv*, 1212.4693. 49

Betancourt, M. and Stein, L. C. (2011). The geometry of Hamiltonian Monte Carlo. *arXiv*, 1112.4118. 49

Girolami, M. and Calderhead, B. (2011). Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2):123–214. 49

Hoffman, M. D. and Gelman, A. (2011). The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *arXiv*, 1111.4246. 45, 46, 49

Hoffman, M. D. and Gelman, A. (2014). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623. 45, 46, 49

Kucukelbir, A., Ranganath, R., Gelman, A., and Blei, D. M. (2015). Automatic variational inference in Stan. *arXiv*, 1506.03431. 67

Neal, R. (2011). MCMC using Hamiltonian dynamics. In Brooks, S., Gelman, A., Jones, G. L., and Meng, X.-L., editors, *Handbook of Markov Chain Monte Carlo*, pages 116–162. Chapman and Hall/CRC. 51, 64

Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer-Verlag, Berlin, second edition. 65, 66